

# Standard ECMA-334

5<sup>th</sup> Edition / December 2017

### C# Language Specification

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001



### ECMA-334

5<sup>th</sup> Edition / December 2017

### C# Language Specification

.

### **Table of Contents**

Foreword	xix
Introduction	хх
1. Scope	1
2. Conformance	3
3. Normative references	5
4. Terms and definitions	7
5. Acronyms and abbreviations	9
6. General description	
7. Lexical structure	
7.1 Programs	
7.2 Grammars	
7.2.1 General	
7.2.2 Grammar notation	
7.2.3 Lexical grammar	
7.2.4 Syntactic grammar	
7.2.5 Grammar ambiguities	
7.3 Lexical analysis	
7.3.1 General	
7.3.2 Line terminators	
7.3.3 Comments	
7.3.4 White space	
7.4 Tokens	
7.4.1 General	
7.4.2 Unicode character escape sequences	
7.4.3 Identifiers	
7.4.4 Keywords	21
7.4.5 Literals	
7.4.5.1 General	
7.4.5.2 Boolean literals	
7.4.5.3 Integer literals	
7.4.5.4 Real literals	
7.4.5.5 Character literals	24
7.4.5.6 String literals	
7.4.5.7 The null literal	
7.4.6 Operators and punctuators	
7.5 Pre-processing directives	
7.5.1 General	
7.5.2 Conditional compilation symbols	
7.5.3 Pre-processing expressions	
7.5.4 Definition directives	
7.5.5 Conditional compilation directives	
7.5.6 Diagnostic directives	
7.5.7 Region directives	

7.5.8 Line directives	
7.5.9 Pragma directives	
8. Basic concepts	
8.1 Application startup	
8.2 Application termination	
8.3 Declarations	
8.4 Members	
8.4.1 General	
8.4.2 Namespace members	
8.4.3 Struct members	
8.4.4 Enumeration members	
8.4.5 Class members	
8.4.6 Interface members	
8.4.7 Array members	
8.4.8 Delegate members	
8.5 Member access	
8.5.1 General	
8.5.2 Declared accessibility	
8.5.3 Accessibility domains	
8.5.4 Protected access	
8.5.5 Accessibility constraints	
8.6 Signatures and overloading	
8.7 Scopes	
8.7.1 General	
8.7.2 Name hiding	
8.7.2.1 General	
8.7.2.2 Hiding through nesting	
8.7.2.3 Hiding through inheritance	
8.8 Namespace and type names	
8.8.1 General	
8.8.2 Ungualified names	54
8.8.3 Fully qualified names	54
8.9 Automatic memory management	55
8.10 Execution order	
9. Types	
9.1 General	
9.2 Reference types	
9.2.1 General	
9.2.2 Class types	
9.2.3 The object type	
9.2.4 The dynamic type	
9.2.5 The string type	
9.2.6 Interface types	
9.2.7 Array types	
9.2.8 Delegate types	
9.3 Value types	
9.3.1 General	
9.3.2 The System.ValueType type	
9.3.3 Default constructors	
9.3.4 Struct types	
J.J.+ JUUCI (JPC)	03

9.3.5 Simple types	
9.3.6 Integral types	64
9.3.7 Floating-point types	65
9.3.8 The decimal type	66
9.3.9 The bool type	66
9.3.10 Enumeration types	66
9.3.11 Nullable value types	67
9.3.12 Boxing and unboxing	67
9.4 Constructed types	67
9.4.1 General	67
9.4.2 Type arguments	68
9.4.3 Open and closed types	68
9.4.4 Bound and unbound types	
9.4.5 Satisfying constraints	
9.5 Type parameters	
9.6 Expression tree types	
9.7 The dynamic type	
10. Variables	
10.1 General	73
10.2 Variable categories	
10.2.1 General	
10.2.2 Static variables	73
10.2.3 Instance variables	73
10.2.3.1 General	
10.2.3.2 Instance variables in classes	74
10.2.3.3 Instance variables in structs	74
10.2.4 Array elements	74
10.2.5 Value parameters	74
10.2.6 Reference parameters	74
10.2.7 Output parameters	75
10.2.8 Local variables	75
10.3 Default values	76
10.4 Definite assignment	76
10.4.1 General	76
10.4.2 Initially assigned variables	77
10.4.3 Initially unassigned variables	77
10.4.4 Precise rules for determining definite assignment	77
10.4.4.1 General	77
10.4.4.2 General rules for statements	78
10.4.4.3 Block statements, checked, and unchecked statements	78
10.4.4.4 Expression statements	78
10.4.4.5 Declaration statements	
10.4.4.6 If statements	79
10.4.4.7 Switch statements	79
10.4.4.8 While statements	
10.4.4.9 Do statements	79
10.4.4.10 For statements	
10.4.4.11 Break, continue, and goto statements	
10.4.4.12 Throw statements	
10.4.4.13 Return statements	
10.4.4.14 Try-catch statements	
,	

10.4.4.15 Try-finally statements	
10.4.4.16 Try-catch-finally statements	
10.4.4.17 Foreach statements	
10.4.4.18 Using statements	
10.4.4.19 Lock statements	
10.4.4.20 Yield statements	
10.4.4.21 General rules for constant expressions	
10.4.4.22 General rules for simple expressions	
10.4.4.23 General rules for expressions with embedded expressions	
10.4.4.24 Invocation expressions and object creation expressions	
10.4.4.25 Simple assignment expressions	
10.4.4.26 && expressions	
10.4.4.27    expressions	
10.4.4.28 ! expressions	
10.4.4.29 ?? expressions	
10.4.4.30 ?: expressions	
10.4.4.31 Anonymous functions	
10.5 Variable references	
10.6 Atomicity of variable references	
11. Conversions	
11.1 General	
11.2 Implicit conversions	
11.2.1 General	
11.2.2 Identity conversion	
11.2.3 Implicit numeric conversions	
11.2.4 Implicit enumeration conversions	
11.2.5 Implicit nullable conversions	
11.2.6 Null literal conversions	
11.2.7 Implicit reference conversions	
11.2.8 Boxing conversions	91
11.2.9 Implicit dynamic conversions	
11.2.10 Implicit constant expression conversions	
11.2.11 Implicit conversions involving type parameters	
11.2.12 User-defined implicit conversions	
11.2.13 Anonymous function conversions and method group conversions	
11.3 Explicit conversions	
11.3.1 General	
11.3.2 Explicit numeric conversions	
11.3.3 Explicit enumeration conversions	
11.3.4 Explicit nullable conversions	
11.3.5 Explicit reference conversions	
11.3.6 Unboxing conversions	
11.3.7 Explicit dynamic conversions	
11.3.8 Explicit conversions involving type parameters	
11.3.9 User-defined explicit conversions	
11.4 Standard conversions	
11.4.1 General	
11.4.2 Standard implicit conversions	
11.4.3 Standard explicit conversions	
11.5 User-defined conversions	
11.5.1 General	

11.5.2 Permitted user-defined conversions	
11.5.3 Evaluation of user-defined conversions	
11.5.4 User-defined implicit conversions	
11.5.5 User-defined explicit conversions	103
11.6 Conversions involving nullable types	
11.6.1 Nullable Conversions	
11.6.2 Lifted conversions	
11.7 Anonymous function conversions	105
11.7.1 General	105
11.7.2 Evaluation of anonymous function conversions to delegate types	
11.7.3 Evaluation of anonymous function conversions to expression tree types	
11.8 Method group conversions	
12. Expressions	
12.1 General	
12.2 Expression classifications	
12.2.1 General	
12.2.2 Values of expressions	
12.3 Static and Dynamic Binding	
12.3.1 General	
12.3.2 Binding-time	
12.3.3 Dynamic binding	
12.3.4 Types of subexpressions	
12.4 Operators	
12.4.1 General	
12.4.2 Operator precedence and associativity	
12.4.3 Operator overloading	
12.4.4 Unary operator overload resolution	
12.4.5 Binary operator overload resolution	
12.4.6 Candidate user-defined operators	
12.4.7 Numeric promotions	
12.4.7.1 General	
12.4.7.2 Unary numeric promotions	
12.4.7.3 Binary numeric promotions	
12.4.8 Lifted operators	
12.5 Member lookup	
12.5.1 General	
12.5.2 Base types	
12.6 Function members	
12.6.1 General	
12.6.2 Argument lists	
12.6.2.1 General	
12.6.2.2 Corresponding parameters	
12.6.2.3 Run-time evaluation of argument lists	
12.6.3 Type inference	
12.6.3.1 General	
12.6.3.2 The first phase	
12.6.3.3 The second phase	
12.6.3.4 Input types	
12.6.3.5 Output types	
12.6.3.6 Dependence	
12.6.3.7 Output type inferences	

12.6.3.8 Explicit parameter type inferences	
12.6.3.9 Exact inferences	
12.6.3.10 Lower-bound inferences	
12.6.3.11 Upper-bound inferences	
12.6.3.12 Fixing	
12.6.3.13 Inferred return type	
12.6.3.14 Type inference for conversion of method groups	
12.6.3.15 Finding the best common type of a set of expressions	
12.6.4 Overload resolution	
12.6.4.1 General	
12.6.4.2 Applicable function member	
12.6.4.3 Better function member	
12.6.4.4 Better conversion from expression	
12.6.4.5 Better conversion from type	
12.6.4.6 Better conversion target	
12.6.4.7 Overloading in generic classes	
12.6.5 Compile-time checking of dynamic member invocation	
12.6.6 Function member invocation	
12.6.6.1 General	
12.6.6.2 Invocations on boxed instances	
12.7 Primary expressions	
12.7.1 General	
12.7.2 Literals	
12.7.3 Simple names	
12.7.3.1 General	
12.7.3.2 Invariant meaning in blocks	
12.7.4 Parenthesized expressions	
12.7.5 Member access	
12.7.5.1 General	
12.7.5.2 Identical simple names and type names	
12.7.6 Invocation expressions	
12.7.6.1 General	
12.7.6.2 Method invocations	
12.7.6.3 Extension method invocations	
12.7.6.4 Delegate invocations	
12.7.7 Element access	
12.7.7.1 General	
12.7.7.2 Array access	
12.7.7.3 Indexer access	
12.7.8 This access	
12.7.9 Base access	
12.7.10 Postfix increment and decrement operators	
12.7.11 The new operator	
12.7.11.1 General	
12.7.11.2 Object creation expressions	
12.7.11.3 Object initializers	
12.7.11.4 Collection initializers	
12.7.11.5 Array creation expressions	
12.7.11.6 Delegate creation expressions	
12.7.11.7 Anonymous object creation expressions	
12.7.12 The typeof operator	

12.7.13 The sizeof operator	164
12.7.14 The checked and unchecked operators	164
12.7.15 Default value expressions	
12.7.16 Anonymous method expressions	167
12.8 Unary operators	167
12.8.1 General	
12.8.2 Unary plus operator	
12.8.3 Unary minus operator	
12.8.4 Logical negation operator	
12.8.5 Bitwise complement operator	
12.8.6 Prefix increment and decrement operators	
12.8.7 Cast expressions	
12.8.8 Await expressions	
12.8.8.1 General	
12.8.8.2 Awaitable expressions	
12.8.8.3 Classification of await expressions	
12.8.8.4 Run-time evaluation of await expressions	
12.9 Arithmetic operators	
12.9.1 General	
12.9.1 Generalization operator	
12.9.3 Division operator	
12.9.4 Remainder operator	
12.9.4 Remainder Operator	
•	
12.9.6 Subtraction operator 12.10 Shift operators	
12.11 Relational and type-testing operators	
12.11.1 General	
12.11.2 Integer comparison operators	
12.11.3 Floating-point comparison operators	
12.11.4 Decimal comparison operators	
12.11.5 Boolean equality operators	
12.11.6 Enumeration comparison operators	
12.11.7 Reference type equality operators	
12.11.8 String equality operators	
12.11.9 Delegate equality operators	
12.11.10 Equality operators between nullable value types and the null literal	
12.11.11 The is operator	
12.11.12 The as operator	
12.12 Logical operators	
12.12.1 General	
12.12.2 Integer logical operators	
12.12.3 Enumeration logical operators	
12.12.4 Boolean logical operators	
12.12.5 Nullable Boolean & and   operators	
12.13 Conditional logical operators	
12.13.1 General	
12.13.2 Boolean conditional logical operators	
12.13.3 User-defined conditional logical operators	
12.14 The null coalescing operator	
12.15 Conditional operator	
12.16 Anonymous function expressions	
· · · · · · · · · · · · · · · · · · ·	

12.16.1 General	
12.16.2 Anonymous function signatures	
12.16.3 Anonymous function bodies	
12.16.4 Overload resolution	
12.16.5 Anonymous functions and dynamic binding	
12.16.6 Outer variables	
12.16.6.1 General	
12.16.6.2 Captured outer variables	
12.16.6.3 Instantiation of local variables	
12.16.7 Evaluation of anonymous function expressions	
12.16.8 Implementation Exmple	
12.17 Query expressions	
12.17.1 General	
12.17.2 Ambiguities in query expressions	
12.17.3 Query expression translation	
12.17.3.1 General	
12.17.3.2 select and group by clauses with continuations	
12.17.3.3 Explicit range variable types	
12.17.3.4 Degenerate query expressions	
12.17.3.5 From, let, where, join and orderby clauses	
12.17.3.6 Select clauses	
12.17.3.7 Group clauses	
12.17.3.8 Transparent identifiers	
12.17.4 The query-expression pattern	
12.18 Assignment operators	
12.18.1 General	
12.18.2 Simple assignment	
12.18.3 Compound assignment	
12.18.4 Event assignment	
12.19 Expression	
12.20 Constant expressions	
12.21 Boolean expressions	
13. Statements	
13.1 General	
13.2 End points and reachability	219
13.3 Blocks	
13.3.1 General	
13.3.2 Statement lists	
13.4 The empty statement	
13.5 Labeled statements	
13.6 Declaration statements	
13.6.1 General	
13.6.2 Local variable declarations	
13.6.3 Local constant declarations	
13.7 Expression statements	
13.8 Selection statements	
13.8.1 General	
13.8.2 The if statement	
13.8.3 The switch statement	
13.9 Iteration statements	230
13.9.1 General	220

13.9.2 The while statement	
13.9.3 The do statement	
13.9.4 The for statement	
13.9.5 The foreach statement	
13.10 Jump statements	
13.10.1 General	
13.10.2 The break statement	
13.10.3 The continue statement	
13.10.4 The goto statement	
13.10.5 The return statement	
13.10.6 The throw statement	
13.11 The try statement	
13.12 The checked and unchecked statements	
13.13 The lock statement	
13.14 The using statement	
13.15 The yield statement	
14. Namespaces	
14.1 General	
14.2 Compilation units	
14.3 Namespace declarations	
14.4 Extern alias directives	
14.5 Using directives	
14.5.1 General	
14.5.2 Using alias directives	
14.5.3 Using namespace directives	
14.6 Namespace member declarations	
14.7 Type declarations	
14.8 Qualified alias member	
14.8.1 General	
14.8.2 Uniqueness of aliases	
15. Classes	
15.1 General	
15.2 Class declarations	
15.2.1 General	
15.2.2 Class modifiers	
15.2.2.1 General	
15.2.2.2 Abstract classes	
15.2.2.3 Sealed classes	
15.2.2.4 Static classes	
15.2.3 Type parameters	
15.2.4 Class base specification	
15.2.4.1 General	
15.2.4.2 Base classes	
15.2.4.3 Interface implementations	
15.2.5 Type parameter constraints	
15.2.6 Class body	
15.2.7 Partial declarations	
15.3 Class members	
15.3.1 General	
15.3.2 The instance type	
<i>//</i>	

15.3.3 Members of constructed types	
15.3.4 Inheritance	
15.3.5 The new modifier	
15.3.6 Access modifiers	
15.3.7 Constituent types	
15.3.8 Static and instance members	
15.3.9 Nested types	
15.3.9.1 General	
15.3.9.2 Fully qualified name	
15.3.9.3 Declared accessibility	
15.3.9.4 Hiding	
15.3.9.5 this access	
15.3.9.6 Access to private and protected members of the containing type	
15.3.9.7 Nested types in generic classes	
15.3.10 Reserved member names	
15.3.10.1 General	
15.3.10.2 Member names reserved for properties	
15.3.10.3 Member names reserved for events	
15.3.10.4 Member names reserved for indexers	
15.3.10.5 Member names reserved for finalizers	
15.4 Constants	
15.5 Fields	
15.5.1 General	
15.5.2 Static and instance fields	
15.5.3 Readonly fields	
15.5.3.1 General	
15.5.3.2 Using static readonly fields for constants	
15.5.3.3 Versioning of constants and static readonly fields	
15.5.4 Volatile fields	
15.5.5 Field initialization	
15.5.6 Variable initializers	
15.5.6.1 General	
15.5.6.2 Static field initialization	
15.5.6.3 Instance field initialization	
15.6 Methods	
15.6.1 General	
15.6.2 Method parameters	
15.6.2.1 General	
15.6.2.2 Value parameters	
15.6.2.3 Reference parameters	
15.6.2.4 Output parameters	
15.6.2.5 Parameter arrays	
15.6.3 Static and instance methods	
15.6.4 Virtual methods	
15.6.5 Override methods	
15.6.6 Sealed methods	
15.6.7 Abstract methods	
15.6.8 External methods	
15.6.9 Partial methods	
15.6.10 Extension methods	
15.6.11 Method body	

15.7 Properties	
15.7.1 General	
15.7.2 Static and instance properties	
15.7.3 Accessors	
15.7.4 Automatically implemented properties	
15.7.5 Accessibility	
15.7.6 Virtual, sealed, override, and abstract accessors	
15.8 Events	
15.8.1 General	
15.8.2 Field-like events	
15.8.3 Event accessors	
15.8.4 Static and instance events	
15.8.5 Virtual, sealed, override, and abstract accessors	
15.9 Indexers	
15.10 Operators	
15.10.1 General	
15.10.2 Unary operators	
15.10.3 Binary operators	
15.10.4 Conversion operators	
15.11 Instance constructors	
15.11.1 General	
15.11.2 Constructor initializers	
15.11.3 Instance variable initializers	
15.11.4 Constructor execution	
15.11.5 Default constructors	
15.12 Static constructors	
15.13 Finalizers	
15.14 Iterators	
15.14.1 General	
15.14.2 Enumerator interfaces	
15.14.3 Enumerable interfaces	
15.14.4 Yield type	
15.14.5 Enumerator objects	
15.14.5.1 General	
15.14.5.2 The MoveNext method	
15.14.5.3 The Current property	
15.14.5.4 The Dispose method	
15.14.6 Enumerable objects	
15.14.6.1 General	
15.14.6.2 The GetEnumerator method	
15.15 Async Functions	
15.15.1 General	
15.15.2 Evaluation of a task-returning async function	
15.15.3 Evaluation of a void-returning async function	
16. Structs	
16.1 General	
16.2 Struct declarations	
16.2.1 General	
16.2.2 Struct modifiers	
16.2.3 Partial modifier	
16.2.4 Struct interfaces	

10.2.9 Struct body	
16.3 Struct members	
16.4 Class and struct differences	
16.4.1 General	
16.4.2 Value semantics	
16.4.3 Inheritance	
16.4.4 Assignment	
16.4.5 Default values	
16.4.6 Boxing and unboxing	
16.4.7 Meaning of this	
16.4.8 Field initializers	
16.4.9 Constructors	
16.4.10 Static constructors	
16.4.11 Automatically implemented properties	
17. Arrays	
17.1 General	
17.2 Array types	
17.2 Analy types	
17.2.1 General	
17.2.2 The System Array type	
17.2.3 Arrays and the generic conection interfaces	
17.5 Array cleation	
•	
17.5 Array members	
17.6 Array covariance	
17.7 Array initializers	
18. Interfaces	
18.1 General 18.2 Interface declarations	
18.1 General	
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> </ul>	
18.1 General 18.2 Interface declarations 18.2.1 General	
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> </ul>	
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> </ul>	
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> </ul>	357 357 357 357 357 358 358 358 358
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> </ul>	357 357 357 357 357 358 358 358 358 358 358 359
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> </ul>	
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> </ul>	357 357 357 357 357 358 358 358 358 358 358 359 359 360
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> </ul>	357 357 357 357 357 358 358 358 358 358 359 359 359 359 360
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> </ul>	357 357 357 357 358 358 358 358 358 358 359 359 359 360 360 360
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> <li>18.4.1 General</li> </ul>	357 357 357 357 357 358 358 358 358 358 359 359 360 360 360 361
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> <li>18.4.1 General</li> <li>18.4.2 Interface methods</li> </ul>	357 357 357 357 358 358 358 358 358 359 359 359 359 360 360 360 361 361
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> <li>18.4.1 General</li> <li>18.4.2 Interface methods</li> <li>18.4.3 Interface properties</li> </ul>	357 357 357 357 358 358 358 358 358 359 359 360 360 360 360 360 361 362 362
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> <li>18.4.1 General</li> <li>18.4.2 Interface methods</li> <li>18.4.3 Interface properties</li> <li>18.4.4 Interface events</li> <li>18.4.5 Interface indexers</li> </ul>	357 357 357 357 357 358 358 358 358 359 359 360 360 360 360 361 362 362 362
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> <li>18.4.1 General</li> <li>18.4.2 Interface methods</li> <li>18.4.3 Interface properties</li> <li>18.4.4 Interface events</li> </ul>	357 357 357 357 358 358 358 358 359 359 359 360 360 360 360 361 362 362 362 362 362 363
<ul> <li>18.1 General</li> <li>18.2 Interface declarations</li> <li>18.2.1 General</li> <li>18.2.2 Interface modifiers</li> <li>18.2.3 Variant type parameter lists</li> <li>18.2.3.1 General</li> <li>18.2.3.2 Variance safety</li> <li>18.2.3.3 Variance conversion</li> <li>18.2.4 Base interfaces</li> <li>18.3 Interface body</li> <li>18.4 Interface members</li> <li>18.4.1 General</li> <li>18.4.2 Interface methods</li> <li>18.4.3 Interface properties</li> <li>18.4.4 Interface events</li> <li>18.4.5 Interface indexers</li> <li>18.4.6 Interface member access</li> </ul>	357 357 357 357 358 358 358 358 359 359 360 360 360 360 360 361 362 362 362 362 362 363 364
<ul> <li>18.1 General.</li> <li>18.2 Interface declarations</li></ul>	357 357 357 357 358 358 358 358 359 359 360 360 360 360 361 362 362 362 362 362 362 362 362 362 363
<ul> <li>18.1 General</li></ul>	357 357 357 357 358 358 358 358 359 359 359 359 360 360 360 360 360 361 362 362 362 362 362 362 362 362 363 364 365
18.1 General18.2 Interface declarations18.2.1 General18.2.2 Interface modifiers18.2.3 Variant type parameter lists18.2.3.1 General18.2.3.2 Variance safety18.2.3.3 Variance conversion18.2.4 Base interfaces18.3 Interface body18.4.1 General18.4.2 Interface members18.4.3 Interface properties18.4.4 Interface methods18.4.5 Interface indexers18.4.6 Interface member access18.5 Qualified interface member names18.6.1 General18.6.2 Explicit interface member implementations	357 357 357 357 358 358 358 358 359 359 359 360 360 360 360 360 360 360 360 361 362 362 362 362 362 362 362 362 362 363 364 365 365
<ul> <li>18.1 General</li></ul>	357 357 357 357 358 358 358 358 359 359 360 360 360 360 360 360 361 362 362 362 362 362 362 362 362 362 363 364 363 364 365 365 365
18.1 General         18.2 Interface declarations         18.2.1 General         18.2.2 Interface modifiers         18.2.3 Variant type parameter lists         18.2.3 Variant type parameter lists         18.2.3 Variant type parameter lists         18.2.3 Variance conversion         18.2.4 Base interfaces         18.3 Interface body         18.4 Interface members         18.4.1 General         18.4.2 Interface methods         18.4.3 Interface properties         18.4.4 Interface member access         18.4.5 Interface indexers         18.4.6 Interface member access         18.4.6 Interface member names         18.5 Qualified interface member names         18.6.1 General         18.6.2 Explicit interface member implementations         18.6.3 Uniqueness of implemented interfaces	357 357 357 357 358 358 358 358 359 359 359 359 360 360 360 360 360 360 361 362 362 362 362 362 362 362 362 362 363 364 365 365 365 365 368

18.6.6 Interface implementation inheritance	
18.6.7 Interface re-implementation	
18.6.8 Abstract classes and interfaces	
19. Enums	
19.1 General	
19.1 General	
19.2 Enum declarations	
19.5 Enum members	
19.5 The System.Enum type 19.6 Enum values and operations	
20. Delegates	
20.1 General	
20.2 Delegate declarations	
20.3 Delegate members	
20.4 Delegate compatibility	
20.5 Delegate instantiation	
20.6 Delegate invocation	
21. Exceptions	
21.1 General	
21.2 Causes of exceptions	
21.3 The System.Exception class	
21.4 How exceptions are handled	
21.5 Common exception classes	
22. Attributes	
22.1 General	
22.2 Attribute classes	
22.2.1 General	
22.2.2 Attribute usage	
22.2.3 Positional and named parameters	
22.2.4 Attribute parameter types	
22.3 Attribute specification	
22.4 Attribute instances	
22.4.1 General	
22.4.2 Compilation of an attribute	
22.4.3 Run-time retrieval of an attribute instance	
22.5 Reserved attributes	
22.5.1 General	
22.5.2 The AttributeUsage attribute	
22.5.3 The Conditional attribute	
22.5.3.1 General	
22.5.3.2 Conditional methods	
22.5.3.3 Conditional attribute classes	
22.5.4 The Obsolete attribute	
22.5.5 Caller-info attributes	
22.5.5.1 General	
22.5.5.2 The CallerLineNumber attribute	
22.5.5.3 The CallerFilePath attribute	
22.5.5.4 The CallerMemberName attribute	
22.6 Attributes for interoperation	

23. Unsafe code	
23.1 General	
23.2 Unsafe contexts	
23.3 Pointer types	
23.4 Fixed and moveable variables	
23.5 Pointer conversions	
23.5.1 General	
23.5.2 Pointer arrays	
23.6 Pointers in expressions	
23.6.1 General	
23.6.2 Pointer indirection	
23.6.3 Pointer member access	
23.6.4 Pointer element access	
23.6.5 The address-of operator	
23.6.6 Pointer increment and decrement	
23.6.7 Pointer arithmetic	
23.6.8 Pointer comparison	
23.6.9 The sizeof operator	
23.7 The fixed statement	
23.8 Fixed-size buffers	
23.8.1 General	
23.8.2 Fixed-size buffer declarations	
23.8.3 Fixed-size buffers in expressions	
23.8.4 Definite assignment checking	
23.9 Stack allocation	
Annex A. Grammar	420
A.1 General	
A.2 Lexical grammar	
A.2.1 Comments	
A.2.2 Tokens	
A.2.3 Keywords	
A.2.4 Operators and punctuators	
A.2.5 Pre-processing directives	
A.3 Syntactic grammar	
A.3.1 Basic concepts	
A.3.2 Types	
A.3.3 Variables	
A.3.4 Expressions	
A.3.5 Statements	
A.3.6 Namespaces	
A.3.7 Classes	
A.3.8 Structs	
A.3.9 Arrays	
A.3.10 Interfaces	
A.3.11 Enums	
A.3.12 Delegates	
A.3.13 Attributes	
A.4 Grammar extensions for unsafe code	
Annex B. Portability issues	
B.1 General	

B.2 Undefined behavior	
B.3 Implementation-defined behavior	
B.4 Unspecified behavior	
B.5 Other Issues	
Annex C. Standard library	
C.1 General	
C.2 Standard Library Types defined in ISO/IEC 23271	
C.3 Standard Library Types not defined in ISO/IEC 23271:2012	
Annex D. Documentation comments	
D.1 General	
D.2 Introduction	
D.3 Recommended tags	
D.3.1 General	
D.3.2 <c></c>	
D.3.3 <code></code>	
D.3.4 <example></example>	
D.3.5 <exception></exception>	
D.3.6 <include></include>	
D.3.7 <list></list>	
D.3.8 <para></para>	
D.3.9 <param/>	
D.3.10 <paramref></paramref>	
D.3.11 <permission></permission>	
D.3.12 <remarks></remarks>	
D.3.13 <returns></returns>	
D.3.14 <see></see>	
D.3.15 <seealso></seealso>	
D.3.16 <summary></summary>	
D.3.17 <typeparam></typeparam>	
D.3.18 <typeparamref></typeparamref>	
D.3.19 <value></value>	
D.4 Processing the documentation file	
D.4.1 General	
D.4.2 ID string format	
D.4.3 ID string examples	
D.5 An example	
D.5.1 C# source code	
D.5.2 Resulting XML	
Annex E. Bibliography	

### Foreword

This specification replaces ECMA-334:2006. Changes from the previous edition include the following:

- Addition of
  - o default and hidden options on the #line preprocessing directive
  - Fixed-size buffers in unsafe code
  - o Automatically implemented properties
  - o Implicitly typed local variables and arrays
  - o Object and collection initializers
  - Anonymous types
  - o Lambda expressions
  - o Expression trees
  - Improved type inference
  - $\circ \quad \text{Extension methods} \quad$
  - Query expressions
  - o Optional parameters
  - o Named arguments
  - o Generic variance
  - o Dynamic binding
  - Asynchronous functions
  - Caller-info attributes
- Removal of
  - o concept of a null type
- Integration of
  - o nullable value types
  - o generic types and functions
  - $\circ$  iterators

### Introduction

This specification is based on a submission from Hewlett-Packard, Intel, and Microsoft, that described 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 run-time 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 (see Annex C).

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 regard to memory and processing power requirements, the language was not intended to compete directly on performance and size with C or assembly language.

## 1. Scope

This specification describes the form and establishes the interpretation of programs written in the C# programming language. It describes

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

This specification does not describe

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

### 2. Conformance

Conformance is of interest to the following audiences:

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

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

The text in this specification that specifies requirements is considered *normative*. All other text in this specification is *informative*; that is, for information purposes only. Unless stated otherwise, all text is normative. Normative text is further broken into *required* and *conditional* categories. *Conditionally normative* text specifies a feature and its requirements where the feature is optional. However, if that feature is provided, its syntax and semantics shall be exactly as specified.

Undefined behavior is indicated in this specification only by the words "undefined behavior."

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

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

A conforming implementation of C# shall provide and support all the types, values, objects, properties, methods, and program syntax and semantics described in the normative (but not the conditionally normative) parts in this specification.

A conforming implementation of C# shall interpret characters in conformance with the Unicode Standard. Conforming implementations shall accept Unicode source files encoded with the UTF-8 encoding form.

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

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

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

A conforming implementation of C# shall be accompanied by a document that defines all implementationdefined characteristics, and all extensions.

A conforming implementation of C# shall support the class library documented in Annex C. This library is included by reference in this specification.

A *conforming program* is one that is acceptable to a conforming implementation. (Such a program is permitted to contain extensions or conditionally normative features.)

### 3. Normative references

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

ISO/IEC 23271:2012, Common Language Infrastructure (CLI), Partition IV: Base Class Library (BCL), Extended Numerics Library, and Extended Array Library.

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

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

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

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989). (This standard is widely known by its U.S. national designation, ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*)

The Unicode Consortium. The Unicode Standard, http://www.unicode.org/standard/standard.html

### 4. Terms and definitions

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

#### 4.1

application

assembly with an entry point

#### 4.2

#### application domain

entity that enables application isolation by acting as a container for application state

#### 4.3

#### argument

expression in the comma-separated list bounded by the parentheses in a method or instance constructor call expression or bounded by the square brackets in an element access expression

#### 4.4

#### assembly

one or more files output by the compiler as a result of program compilation

#### 4.5

#### behavior

external appearance or action

#### 4.6

#### behavior, implementation-defined

unspecified behavior where each implementation documents how the choice is made

#### 4.7

#### behavior, undefined

behavior, upon use of a non-portable or erroneous construct or of erroneous data, for which this specification imposes no requirements

#### 4.8

#### behavior, unspecified

behavior where this specification provides two or more possibilities and imposes no further requirements on which is chosen in any instance

#### 4.9

character (when used without a qualifier)

a) In the context of a non-Unicode encoding – the meaning of character in that encoding; or

b) In the context of a character literal or a value of type char – a Unicode code point in the range U+0000 to U+FFFF (including surrogate code points), that is a UTF-16 code unit; or

c) Otherwise – a Unicode code point

#### 4.10

#### class library

assembly that can be used by other assemblies

#### 4.11

#### diagnostic message

message belonging to an implementation-defined subset of the implementation's output messages

#### 4.12

#### error, compile-time

error reported during program translation

#### 4.13

#### exception

exceptional condition reported during program execution

#### 4.14

#### implementation

particular set of software (running in a particular translation environment under particular control options) that performs translation of programs for, and supports execution of methods in, a particular execution environment

#### 4.15

#### namespace

logical organizational system grouping related program elements

#### 4.16

#### parameter

variable declared as part of a method, instance constructor, operator, or indexer definition, which acquires a value on entry to that function member

#### 4.17

#### program

one or more source files that are presented to the compiler

#### 4.18

#### program, valid

C# program constructed according to the syntax rules and diagnosable semantic rules

#### 4.19

#### program instantiation

execution of an application

#### 4.20

#### source file

ordered sequence of Unicode characters

#### 4.21

#### unsafe code

code that is permitted to perform such lower-level operations as declaring and operating on pointers, performing conversions between pointers and integral types, and taking the address of variables

#### 4.22

#### warning, compile-time

informational message reported during program translation, which is intended to identify a potentially questionable usage of a program element

### 5. Acronyms and abbreviations

#### This clause is informative.

The following acronyms and abbreviations are used throughout this specification:

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+0023).

The following types appear throughout this specification. The full names of those types, including the global namespace qualifier are listed below for reference. Throughout this specification, these types will appear as the fully qualified name, omitting the global namespace qualifier, or as a simple unqualified type name, omitting the namespace. For example, the type ICollection<T>, when used in this specification, always means the type global::System.Collections.Generic.ICollection<T>.

- global::System.Action
- global::System.ArgumentException
- global::System.ArithmeticException
- global::System.Array
- global::System.ArrayTypeMisMatchException
- global::System.Attribute
- global::System.AttributeTargets
- global::System.AttributeUsageAttribute
- global::System.Boolean
- global::System.Byte
- global::System.Char
- global::System.Collections.Generic.ICollection<T>
- global::System.Collections.Generic.IEnumerable<T>
- global::System.Collections.Generic.IEnumerator<T>
- global::System.Collections.Generic.IList<T>
- global::System.Collections.Generic.IReadonlyCollection<out T>
- global::System.Collections.Generic.IReadOnlyList<out T>
- global::System.Collections.ICollection
- global::System.Collections.IEnumerable
- global::System.Collections.IList
- global::System.Collections.IEnumerator
- global::System.Decimal

- global::System.Delegate
- global::System.Diagnostics.ConditionalAttribute
- global::System.DivideByZeroException
- global::System.Double
- global::System.Enum
- global::System.Exception
- global::System.GC
- global::System.ICollection
- global::System.IDisposable
- global::System.IEnumerable
- global::System.IEnumerable<out T>
- global::System.IList
- global::System.IndexOutOfRangeException
- global::System.Int16
- global::System.Int32
- global::System.Int64
- global::System.IntPtr
- global::System.InvalidCastException
- global::System.InvalidOperationException
- global::System.Linq.Expressions.Expression<TDelegate>
- global::System.MemberInfo
- global::System.NotSupportedException
- global::System.Nullable<T>
- global::System.NullReferenceException
- global::System.Object
- global::System.ObsoleteAttribute
- global::System.OutOfMemoryException
- global::System.OverflowException
- global::System.Runtime.CompilerServices.CallerFileAttribute
- global::System.Runtime.CompilerServices.CallerLineNumberAttribute
- global::System.Runtime.CompilerServices.CallerMemberNameAttribute
- global::System.Runtime.CompilerServices.ICriticalNotifyCompletion
- global::System.Runtime.CompilerServices.IndexerNameAttribute
- global::System.Runtime.CompilerServices.INotifyCompletion
- global::System.Runtime.CompilerServices.TaskAwaiter
- global::System.Runtime.CompilerServices.TaskAwaiter<T>
- global::System.SByte
- global::System.Single
- global::System.StackOverflowException
- global::System.String
- global::System.SystemException
- global::System.Threading.Monitor
- global::System.Threading.Tasks.Task
- global::System.Threading.Tasks.Task<TResult>
- global::System.Type
- global::System.TypeInializationException
- global::System.UInt16
- global::System.UInt32
- global::System.UInt64
- global::System.UIntPtr
- global::System.ValueType

#### End of informative text.

### 6. General description

#### This text is informative.

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

This standard is divided into the following subdivisions:

- 1. Front matter (clauses 1–6);
- 2. The language syntax, constraints, and semantics (clauses 7–23);
- 3. Annexes

Examples are provided to illustrate possible forms of the constructions described. References are used to refer to related clauses. Notes are provided to give advice or guidance to implementers or programmers. Annexes provide additional information and summarize the information contained in this specification.

Clauses 1–4, part of Clause 6, Clauses 7–22, the beginning of Clause 23, and most of Annex D form a normative part of this standard. With the exception of the beginning, all of Clause 23 is conditionally normative. The Foreword, Introduction, Clause 5, part of Clause 6, Annexes A, B, C, part of Annexes D, E, and F, notes, and examples are informative.

#### End of informative text.

Informative text is indicated in the following ways:

- 1. Whole or partial clauses or annexes delimited by "This clause/text is informative" and "End of informative text".
- 2. [*Example*: The following example ... code fragment, possibly with some narrative ... end example]
- 3. [Note: narrative ... end note]

All text not marked as being informative is normative.

### 7. Lexical structure

#### 7.1 Programs

A C# *program* consists of one or more *source files*, known formally as *compilation units* (§14.2). 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 shall 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 can choose to accept and transform additional character encoding schemes (such as UTF-16, UTF-32, or non-Unicode character mappings).

[*Note*: The handling of the Unicode NULL character (U+0000) is implementation-specific. It is strongly recommended that developers avoid using this character in their source code, for the sake of both portability and readability. When the character is required within a character or string literal, the escape sequences \0 or \u0000 may be used instead. *end note*]

[*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 (U+000D) followed by Unicode line-feed character (U+000A). For the most part this transformation will have no visible effects; however, it will affect the interpretation of verbatim string literal tokens (§7.4.5.6). 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*]

#### 7.2 Grammars

#### 7.2.1 General

This specification presents the syntax of the C# programming language using two grammars. The *lexical grammar* (§7.2.2) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The *syntactic grammar* (§7.2.4) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

#### 7.2.2 Grammar notation

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

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by one or two colons. One colon is used for a production in the syntactic grammar, two colons for a

production in the lexical grammar. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. [*Example*: The production:

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

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

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. [*Example*: The production:

statement-list: statement statement-list statement

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

A subscripted suffix "opt" is used to indicate an optional symbol. [Example: The production:

```
block:
    { statement-list<sub>opt</sub> }
```

is shorthand for:

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

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

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

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

is shorthand for:

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

end example]

All terminal characters are to be understood as the appropriate Unicode character from the range U+0020 to U+007F, as opposed to any similar-looking characters from other Unicode character ranges.

#### 7.2.3 Lexical grammar

The lexical grammar of C# is presented in §7.3, §7.4, and §7.5. 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 (§7.4), white space (§7.3.4), comments (§7.3.3), and pre-processing directives (§7.5).

Every source file in a C# program shall conform to the *input* production of the lexical grammar (§7.3.1).

# 7.2.4 Syntactic grammar

The syntactic grammar of C# is presented in the clauses, subclauses, and annexes that follow this subclause. 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 shall conform to the *compilation-unit* production (§14.2) of the syntactic grammar.

# 7.2.5 Grammar ambiguities

The productions for *simple-name* (§12.7.3) and *member-access* (§12.7.5) can give rise to ambiguities in the grammar for expressions. [*Example*: The statement:

could be interpreted as a call to F with two arguments, G < A and B > (7). Alternatively, it could be interpreted as a call to F with one argument, which is a call to a generic method G with two type arguments and one regular argument. *end example*]

If a sequence of tokens can be parsed (in context) as a *simple-name* (§12.7.3), *member-access* (§12.7.5), or *pointer-member-access* (§23.6.3) ending with a *type-argument-list* (§9.4.2), the token immediately following the closing > token is examined. If it is one of

( ) ] : ; , . ? == !=

then the *type-argument-list* is retained as part of the *simple-name, member-access*, or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered part of the *simple-name, member-access*, or *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. [*Note*: These rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§8.8). *end note*] [*Example*: The statement:

F(G<A, B>(7));

will, according to this rule, be interpreted as a call to F with one argument, which is a call to a generic method G with two type arguments and one regular argument. The statements

F(G<A, B>7); F(G<A, B>>7);

will each be interpreted as a call to F with two arguments. The statement

```
x = F < A > + y;
```

will be interpreted as a less-than operator, greater-than operator and unary-plus operator, as if the statement had been written x = (F < A) > (+y), instead of as a *simple-name* with a *type-argument-list* followed by a binary-plus operator. In the statement

x = y is C<T> && z;

the tokens C<T> are interpreted as a *namespace-or-type-name* with a *type-argument-list* due to being on the right-hand side of the is operator (§12.11.1). Because C<T> parses as a *namespace-or-type-name*, not a *simple-name*, *member-access*, or *pointer-member-access*, the above rule does not apply, and it is considered to have a *type-argument-list* regardless of the token that follows. *end example*]

# 7.3 Lexical analysis

# 7.3.1 General

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

input:: input-section<sub>opt</sub> input-section:: input-section-part input-section input-section-part

input-section-part:: input-elements<sub>opt</sub> new-line pp-directive

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

input-element:: whitespace comment token

Five basic elements make up the lexical structure of a C# source file: Line terminators ( $\S7.3.2$ ), white space (\$7.3.4), comments (\$7.3.3), tokens (\$7.4), and pre-processing directives (\$7.5). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program (\$7.2.4), except in the case of a > token being combined with another token to form a single operator (\$7.4.6).

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

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

## 7.3.2 Line terminators

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

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

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

- If the last character of the source file is a Control-Z character (U+001A), this character is deleted.
- A carriage-return character (U+000D) is added to the end of the source file if that source file is nonempty and if the last character of the source file is not a carriage return (U+000D), a line feed (U+000A), a next line character (U+0085), a line separator (U+2028), or a paragraph separator (U+2029). [Note: The additional carriage-return allows a program to end in a *pp-directive* (§7.5) that does not have a terminating *new-line. end note*]

## 7.3.3 Comments

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

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

includes a delimited comment. end example]

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

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

shows several single-line comments. end example]

```
comment::
   single-line-comment
   delimited-comment
single-line-comment::
   // input-charactersopt
input-characters::
   input-character
   input-characters input-character
input-character::
   Any Unicode character except a new-line-character
new-line-character::
   Carriage return character (U+000D)
   Line feed character (U+000A)
   Next line character (U+0085)
   Line separator character (U+2028)
   Paragraph separator character (U+2029)
delimited-comment::
   /* delimited-comment-text<sub>ont</sub> asterisks /
delimited-comment-text::
   delimited-comment-section
   delimited-comment-text delimited-comment-section
delimited-comment-section::
   asterisksopt not-slash-or-asterisk
asterisks::
    *
   asterisks *
not-slash-or-asterisk::
   Any Unicode character except / or *
```

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

Comments are not processed within character and string literals.

[*Note*: These rules must be interpreted carefully. For instance, in the example below, the delimited comment that begins before A ends between B and C(). The reason is that

// B \*/ C();

is not actually a single-line comment, since // has no special meaning within a delimited comment, and so \*/ does have its usual special meaning in that line.

Likewise, the delimited comment starting before D ends before E. The reason is that "D \*/" is not actually a string literal, since it appears inside a delimited comment.

A useful consequence of /\* and \*/ having no special meaning within a single-line comment is that a block of source code lines can be commented out by putting // at the beginning of each line. In general it does not work to put /\* before those lines and \*/ after them, as this does not properly encapsulate delimited comments in the block, and in general may completely change the structure of such delimited comments.

Example code:

```
static void Main() {
    /* A
    // B */ C();
    Console.WriteLine(/* "D */ "E");
}
```

end note]

#### 7.3.4 White space

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

whitespace:: whitespace-character whitespace whitespace-character

whitespace-character::

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

## 7.4 Tokens

## 7.4.1 General

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

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

# 7.4.2 Unicode character escape sequences

A Unicode escape sequence represents a Unicode code point. Unicode escape sequences are processed in identifiers (§7.4.3), character literals (§7.4.5.5), and regular string literals (§7.4.5.6). A Unicode escape sequence is not processed in any other location (for example, to form an operator, punctuator, or keyword).

unicode-escape-sequence::

\u hex-digit hex-digit hex-digit hex-digit \U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

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

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

[Example: The example

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

shows several uses of  $\0$ , which is the escape sequence for the letter "f". The program is equivalent to

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

end example]

## 7.4.3 Identifiers

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

identifier:: available-identifier @ identifier-or-keyword available-identifier:: An identifier-or-keyword that is not a keyword identifier-or-keyword:: identifier-start-character identifier-part-characters<sub>opt</sub> identifier-start-character:: letter-character

letter-character underscore-character underscore-character::

- \_ (the underscore character U+005F)
- A unicode-escape-sequence representing the character U+005F

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

identifier-part-character::

letter-character decimal-digit-character connecting-character combining-character formatting-character

letter-character::

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

combining-character::

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

decimal-digit-character::

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

connecting-character::

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

formatting-character::

A Unicode character of the class Cf

A unicode-escape-sequence representing a character of the class Cf

[*Note*: For information on the Unicode character classes mentioned above, see *The Unicode Standard. end note*]

[Example: Examples of valid identifiers include "identifier1", "\_identifier2", and "@if". end example]

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

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

[Example: The example:

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

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

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

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

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

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

## 7.4.4 Keywords

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

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

A **contextual keyword** is an identifier-like sequence of characters that has special meaning in certain contexts, but is not reserved, and can be used as an identifier outside of those contexts as well as when prefaced by the @ character.

contextual-keyword: one of the following identifiers						
add	alias	ascending	async	await		
by	descending	dynamic	equals	from		
get	global	group	into	join		
let	orderby	partial	remove	select		
set	value	var	where	yield		

In most cases, the syntactic location of contextual keywords is such that they can never be confused with ordinary identifier usage. For example, within a property declaration, the "get" and "set" identifiers have special meaning (§15.7.3). An identifier other than get or set is never permitted in these locations, so this use does not conflict with a use of these words as identifiers.

In certain cases the grammar is not enough to distinguish contextual keyword usage from identifiers. In all such cases it will be specified how to disambiguate between the two. For example, the contextual keyword var in implicitly typed local variable declarations (§13.6.2) might conflict with a declared type called var, in which case the declared name takes precedence over the use of the identifier as a contextual keyword.

Another example such disambiguation is the contextual keyword await (§12.8.8.1), which is considered a keyword only when inside a method declared async, but can be used as an identifier elsewhere.

Just as with keywords, contextual keywords can be used as ordinary identifiers by prefixing them with the @ character.

[*Note*: When used as contextual keywords, these identifiers cannot contain unicode-escape-sequences. *end note*].

## 7.4.5 Literals

#### 7.4.5.1 General

A *literal* (§12.7.2) is a source code representation of a value.

literal::

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

#### 7.4.5.2 Boolean literals

There are two Boolean literal values: true and false.

boolean-literal:: true false

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

#### 7.4.5.3 Integer literals

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

integer-literal:: decimal-integer-literal hexadecimal-integer-literal decimal-integer-literal:: decimal-digits integer-type-suffix<sub>opt</sub> decimal-digits:: decimal-digit decimal-digits decimal-digit decimal-digit:: one of 0 1 2 3 4 5 6 7 8 9 integer-type-suffix:: one of U u L ] UL U] uL u] LU Lu 10 lu hexadecimal-integer-literal:: 0x hex-digits integer-type-suffix<sub>opt</sub> 0X hex-digits integer-type-suffix<sub>opt</sub>

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

The type of an integer literal is determined as follows:

- If the literal has no suffix, it has the first of these types in which its value can be represented: int, uint, long, ulong.
- If the literal is suffixed by U or u, it has the first of these types in which its value can be represented: uint, ulong.
- If the literal is suffixed by L or 1, it has the first of these types in which its value can be represented: long, ulong.
- If the literal is suffixed by UL, Ul, uL, ul, LU, Lu, lU, or lu, it is of type ulong.

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

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

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

- When an *integer-literal* representing the value 2147483648 (2<sup>31</sup>) and no *integer-type-suffix* appears as the token immediately following a unary minus operator token (§12.8.3), the result (of both tokens) is a constant of type int with the value -2147483648 (-2<sup>31</sup>). In all other situations, such an *integer-literal* is of type uint.
- When an *integer-literal* representing the value 9223372036854775808 (2<sup>63</sup>) and no *integer-type-suffix* or the *integer-type-suffix* L or 1 appears as the token immediately following a unary minus operator token (§12.8.3), the result (of both tokens) is a constant of type long with the value -9223372036854775808 (-2<sup>63</sup>). In all other situations, such an *integer-literal* is of type ulong.

#### 7.4.5.4 Real literals

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

```
real-literal::
```

```
decimal-digits . decimal-digits exponent-part<sub>opt</sub> real-type-suffix<sub>opt</sub>
. decimal-digits exponent-part<sub>opt</sub> real-type-suffix<sub>opt</sub>
decimal-digits exponent-part real-type-suffix<sub>opt</sub>
decimal-digits real-type-suffix
```

exponent-part::

```
e sign<sub>opt</sub> decimal-digits
E sign<sub>opt</sub> decimal-digits
sign:: one of
+ -
real-type-suffix:: one of
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:

• A real literal suffixed by F or f is of type float. [*Example*: The literals 1f, 1.5f, 1e10f, and 123.456F are all of type float. *end example*]

- A real literal suffixed by D or d is of type double. [*Example*: The literals 1d, 1.5d, 1e10d, and 123.456D are all of type double. *end example*]
- A real literal suffixed by M or m is of type decimal. [Example: The literals 1m, 1.5m, 1e10m, and 123.456M are all of type decimal. end example] This literal is converted to a decimal value by taking the exact value, and, if necessary, rounding to the nearest representable value using banker's rounding (§9.3.8). Any scale apparent in the literal is preserved unless the value is rounded. [Note: Hence, the literal 2.900m will be parsed to form the decimal with sign 0, coefficient 2900, and scale 3. end note]

If the magnitude of the specified literal is too large to be represented in the indicated type, a compile-time error occurs. [*Note*: In particular, a *real-literal* will never produce a floating-point infinity. A non-zero *real-literal* may, however, be rounded to zero. *end note*]

The value of a real literal of type float or double is determined by using the IEC 60559 "round to nearest" mode with ties broken to "even" (a value with the least-significant-bit zero), and all digits considered significant.

[*Note*: In a real literal, decimal digits are always required after the decimal point. For example, 1.3F is a real literal but 1.F is not. *end note*]

# 7.4.5.5 Character literals

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

```
character-literal::

' character '

character::

single-character

simple-escape-sequence

hexadecimal-escape-sequence

unicode-escape-sequence

single-character::

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

simple-escape-sequence:: one of

\' \'' \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence::

\x hex-digit hex-digit_opt hex-digit_opt
```

[*Note*: A character that follows a backslash character (\) in a *character* shall 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*]

[*Note*: The use of the x *hexadecimal-escape-sequence* production can be error-prone and hard to read due to the variable number of hexadecimal digits following the x. For example, in the code:

string good = "x9Good text";

string bad = "\x9Bad text";

it might appear at first that the leading character is the same (U+0009, a tab character) in both strings. In fact the second string starts with U+9BAD as all three letters in the word "Bad" are valid hexadecimal digits. As a matter of style, it is recommended that x is avoided in favour of either specific escape sequences (t in this example) or the fixed-length u escape sequence. *end note*]

A hexadecimal escape sequence represents a single Unicode UTF-16 code unit, 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 escape sequence (§7.4.2) in a character literal shall be in the range U+0000 to U+FFFF.

Unicode code Escape **Character name** sequence point \' U+0027 Single quote ... U+0022 Double quote  $\backslash \backslash$ U+005C Backslash \0 U+0000 Null ∖a U+0007 Alert \b U+0008 Backspace \f U+000C Form feed U+000A \n New line \r U+000D Carriage return \t U+0009 Horizontal tab \v U+000B Vertical tab

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

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

## 7.4.5.6 String literals

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

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

```
string-literal::
    regular-string-literal
    verbatim-string-literal
regular-string-literal-characters<sub>opt</sub> "
regular-string-literal-characters::
    regular-string-literal-character
    regular-string-literal-characters regular-string-literal-character
regular-string-literal-character::
    single-regular-string-literal-character
    simple-escape-sequence
    hexadecimal-escape-sequence
single-regular-string-literal-character::
```

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

```
ECMA-334
```

```
verbatim-string-literal::

@" verbatim-string-literal-characters<sub>opt</sub> "
```

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

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

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

quote-escape-sequence::

[Example: The example

```
string a = "Happy birthday, Joel";
                                                       // Happy birthday, Joel
string b = @"Happy birthday, Joel";
                                                        // Happy birthday, Joel
string c = "hello \t world"
                                                        // hello
                                                                         world
string d = @"hello \t world";
                                                       // hello \t world
string e = "Joe said \"Hello\" to me";
string f = @"Joe said ""Hello"" to me";
                                                       // Joe said "Hello" to me
                                                       // Joe said "Hello" to me
string g = "\\\\server\\share\\file.txt"; // \\server\share\file.txt
string h = @"\\server\share\file.txt"; // \\server\share\file.txt
string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three":
```

shows a variety of string literals. The last string literal, j, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as new line characters, are preserved verbatim, and each pair of double-quote characters is replaced by one such character. *end example*]

[*Note*: Any line breaks within verbatim string literals are part of the resulting string. If the exact characters used to form line breaks are semantically relevant to an application, any tools that translate line breaks in source code to different formats (between "\n" and "\r\n", for example) will change application behavior. Developers should be careful in such situations. *end note*]

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

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

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (§12.11.8), appear in the same assembly, these string literals refer to the same string instance. [*Example*: For instance, the output produced by

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

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

## 7.4.5.7 The null literal

null-literal:: null

A *null-literal* represents a null value. It does not have a type, but can be converted to any reference type or nullable value type through a null literal conversion (§11.2.6)."

#### 7.4.6 Operators and punctuators

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

operator-or-punctuator:: one of ( { } Γ ) ] 5 ; \* \_ / % & ٨ 1 ~ + ? ?? && < > :: \_\_\_ = ++ \*= != /= %= -> == <= >= += -= &= |=  $\wedge =$ << <<= right-shift:: > > right-shift-assignment:: > >=

right-shift is made up of the two tokens > and >. Similarly, right-shift-assignment is made up of the two tokens > and >=. Unlike other productions in the syntactic grammar, no characters of any kind (not even whitespace) are allowed between the two tokens in each of these productions. These productions are treated specially in order to enable the correct handling of type-parameter-lists (§15.2.3). [Note: Prior to the addition of generics to C#, >> and >>= were both single tokens. However, the syntax for generics uses the < and > characters to delimit type parameters and type arguments. It is often desirable to use nested constructed types, such as List<Dictionary<string, int>>. Rather than requiring the programmer to separate the > and > by a space, the definition of the two operator-or-punctuators was changed. end note]

# 7.5 Pre-processing directives

## 7.5.1 General

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

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

The following pre-processing directives are available:

• #define and #undef, which are used to define and undefine, respectively, conditional compilation symbols (§7.5.4).

- #if, #elif, #else, and #endif, which are used to skip conditionally sections of source code (§7.5.5).
- #line, which is used to control line numbers emitted for errors and warnings (§7.5.8).
- #error, which is used to issue errors (§7.5.6).
- #region and #endregion, which are used to explicitly mark sections of source code (§7.5.7).
- #pragma, which is used to specify optional contextual information to a compiler (§7.5.9).

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

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

Pre-processing directives are not tokens and are not part of the syntactic grammar of C#. However, preprocessing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of a C# program. [*Example*: When compiled, the program

```
#define A
#undef B
class C
{
#if A
void F() {}
#else
void G() {}
#endif
#if B
void H() {}
#else
void I() {}
#endif
```

results in the exact same sequence of tokens as the program

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

Thus, whereas lexically, the two programs are quite different, syntactically, they are identical. *end example*]

# 7.5.2 Conditional compilation symbols

The conditional compilation functionality provided by the #if, #elif, #else, and #endif directives is controlled through pre-processing expressions (§7.5.3) and conditional compilation symbols.

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

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

- Each unicode-escape-sequence is transformed into its corresponding Unicode character.
- Any *formatting-characters* are removed.

A conditional compilation symbol has two possible states: *defined* or *undefined*. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a #define directive is processed, the conditional compilation symbol named in that directive becomes defined in that

source file. The symbol remains defined until a #undef directive for that same symbol is processed, or until the end of the source file is reached. An implication of this is that #define and #undef directives in one source file have no effect on other source files in the same program.

When referenced in a pre-processing expression (§7.5.3), a defined conditional compilation symbol has the Boolean value true, and an undefined conditional compilation symbol has the Boolean value false. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the value false.

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

## 7.5.3 Pre-processing expressions

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

pp-expression:: whitespace<sub>opt</sub> pp-or-expression whitespace<sub>opt</sub> pp-or-expression:: pp-and-expression pp-or-expression whitespace<sub>opt</sub> || whitespace<sub>opt</sub> pp-and-expression pp-and-expression:: pp-equality-expression pp-and-expression whitespace<sub>opt</sub> && whitespace<sub>opt</sub> pp-equality-expression pp-equality-expression:: pp-unary-expression pp-equality-expression whitespace<sub>opt</sub> == whitespace<sub>opt</sub> pp-unary-expression pp-equality-expression whitespace<sub>opt</sub> != whitespace<sub>opt</sub> pp-unary-expression pp-unary-expression:: pp-primary-expression ! whitespace<sub>opt</sub> pp-unary-expression pp-primary-expression:: true false conditional-symbol ( whitespace<sub>opt</sub> pp-expression whitespace<sub>opt</sub> )

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

Evaluation of a pre-processing expression always yields a Boolean value. The rules of evaluation for a preprocessing expression are the same as those for a constant expression (§12.20), except that the only userdefined entities that can be referenced are conditional compilation symbols.

## 7.5.4 Definition directives

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

pp-declaration:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> define whitespace conditional-symbol pp-new-line whitespace<sub>opt</sub> # whitespace<sub>opt</sub> undef whitespace conditional-symbol pp-new-line pp-new-line:: whitespace<sub>opt</sub> single-line-comment<sub>opt</sub> new-line The processing of a **#define** directive causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of a **#undef** directive causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.

Any #define and #undef directives in a source file shall occur before the first *token* (§7.4) in the source file; otherwise a compile-time error occurs. In intuitive terms, #define and #undef directives shall precede any "real code" in the source file.

[Example: The example:

```
#define Enterprise
#if Professional || Enterprise
#define Advanced
#endif
namespace Megacorp.Data
{
    if Advanced
        class PivotTable {...}
#endif
}
```

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

end example]

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

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

## end example]

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

#define A #define A

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

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

#define A #undef A #undef A

end example]

# 7.5.5 Conditional compilation directives

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

pp-conditional:: pp-if-section pp-elif-sections <sub>opt</sub> pp-else-section <sub>opt</sub> pp-endif
pp-if-section:: whitespace <sub>opt</sub> # whitespace <sub>opt</sub> if whitespace pp-expression pp-new-line conditional-section <sub>opt</sub>
pp-elif-sections:: pp-elif-section pp-elif-sections pp-elif-section
pp-elif-section:: whitespace <sub>opt</sub> # whitespace <sub>opt</sub> elif whitespace pp-expression pp-new-line conditional-section <sub>opt</sub>
pp-else-section:: whitespace <sub>opt</sub> # whitespace <sub>opt</sub> else pp-new-line conditional-section <sub>opt</sub>
pp-endif:: whitespace <sub>opt</sub> # whitespace <sub>opt</sub> endif pp-new-line
conditional-section:: input-section skipped-section
skipped-section:: skipped-section-part skipped-section skipped-section-part
skipped-section-part:: skipped-characters <sub>opt</sub> new-line pp-directive
skipped-characters:: whitespace <sub>opt</sub> not-number-sign input-characters <sub>opt</sub>
not-number-sign:: Any input-character except #

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

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

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

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

The remaining *conditional-sections*, if any, are processed as *skipped-sections*: except for pre-processing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from the source code in the section; and pre-processing directives in the section shall be lexically correct but are not otherwise processed. Within a *conditional-section* that is being processed as a *skipped-section*,

any nested *conditional-sections* (contained in nested #if...#endif and #region...#endregion constructs) are also processed as *skipped-sections*.

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

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

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

```
#define Debug // Debugging on
class PurchaseTransaction
{
    void Commit() {
    #if Debug
        CheckConsistency();
#else
        /* Do something else
#endif
    }
    ...
}
```

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

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

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
    #if Debug
        world
    #else
        Nebraska
    #endif
        ");
    }
}
results in the output:
```

hello, #if Debug

```
world
#else
Nebraska
#endif
```

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

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

always produces the same token stream (class Q { }), regardless of whether 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*]

#### 7.5.6 Diagnostic directives

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

```
pp-diagnostic::
    whitespace<sub>opt</sub> # whitespace<sub>opt</sub> error pp-message
    whitespace<sub>opt</sub> # whitespace<sub>opt</sub> warning pp-message
pp-message::
    new-line
    whitespace input-characters<sub>opt</sub> new-line
```

[Example: The example

```
#if Debug && Retail
    #error A build can't be both debug and retail
#endif
class Test {...}
```

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

# 7.5.7 Region directives

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

```
pp-region::
    pp-start-region conditional-section<sub>opt</sub> pp-end-region
pp-start-region::
    whitespace<sub>opt</sub> # whitespace<sub>opt</sub> region pp-message
pp-end-region::
    whitespace<sub>opt</sub> # whitespace<sub>opt</sub> endregion pp-message
```

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

The lexical processing of a region:

#region

#endregion

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

#if true

#endif

# 7.5.8 Line directives

Line directives may be used to alter the line numbers and source file names that are reported by the compiler in output such as warnings and errors. These values are also used by caller-info attributes (§22.5.5).

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

```
pp-line::
whitespace<sub>opt</sub> # whitespace<sub>opt</sub> line whitespace line-indicator pp-new-line
line-indicator::
decimal-digits whitespace file-name
decimal-digits
default
hidden
file-name::
" file-name-characters "
file-name-characters:
file-name-character
file-name-character
file-name-character file-name-character
```

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

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

A #line hidden directive has no effect on the file and line numbers reported in error messages, or produced by use of CallerLineNumberAttribute (§22.5.5.2). It is intended to affect source level debugging tools so that, when debugging, all lines between a #line hidden directive and the subsequent #line directive (that is not #line hidden) have no line number information, and are skipped entirely when stepping through code.

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

# 7.5.9 Pragma directives

The #pragma preprocessing directive is used to specify contextual information to a compiler. [*Note*: For example, a compiler might provide #pragma directives that

- Enable or disable particular warning messages when compiling subsequent code.
- Specify which optimizations to apply to subsequent code.
- Specify information to be used by a debugger.

end note]

pp-pragma:: whitespaceopt # whitespace<sub>opt</sub> pragma pp-pragma-text pp-pragma-text:: new-line whitespace input-characters<sub>opt</sub> new-line The *input-characters* in the *pp-pragma-text* are interpreted by the compiler in an implementation-defined manner. The information supplied in a #pragma directive shall not change program semantics. A #pragma directive shall only change compiler behavior that is outside the scope of this language specification. If the compiler cannot interpret the *input-characters*, the compiler can produce a warning; however, it shall not produce a compile-time error.

[*Note: pp-pragma-text* can contain arbitrary text; specifically, it need not contain well-formed tokens. *end note*]

# 8. Basic concepts

# 8.1 Application startup

A program may be compiled either as a *class library* to be used as part of other applications, or as an *application* that may be started directly. The mechanism for determining this mode of compilation is implementation-specific and external to this specification.

A program compiled as an application shall contain at least one method qualifying as an entry point by satisfying the following requirements:

- It shall have the name Main.
- It shall be static.
- It shall not be generic.
- It shall be declared in a non-generic type. If the type declaring the method is a nested type, none of its enclosing types may be generic.
- It shall not have the async modifier.
- The return type shall be void or int.
- It shall not be a partial method (§15.6.9) without an implementation.
- The formal parameter list shall either be empty, or have a single value parameter of type string[].

If more than one method qualifying as an entry point is declared within a program, an external mechanism may be used to specify which method is deemed to be the actual entry point for the application. It is a compile-time error for a program to be compiled as an application without exactly one entry point. A program compiled as a class library may contain methods that would qualify as application entry points, but the resulting library has no entry point.

Ordinarily, the declared accessibility (§8.5.2) of a method is determined by the access modifiers (§15.3.6) specified in its declaration, and similarly the declared accessibility of a type is determined by the access modifiers specified in its declaration. In order for a given method of a given type to be callable, both the type and the member shall be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility of its enclosing type declarations.

When an application is run, a new *application domain* is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain. An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same types loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

Application startup occurs when the execution environment calls the application's entry point. If the entry point declares a parameter, then during application startup, the implementation shall ensure that the initial value of parameter is a non-null reference to a string array. This array shall consist of non-null references to strings, called application parameters, which are given implementation-defined values by the host environment prior to application startup. The intent is to supply to the application information determined prior to application startup from elsewhere in the hosted environment. [*Note*: On systems

supporting a command line, application parameters correspond to what are generally known as commandline arguments. *end note*]

If the entry point's return type is int rather than void, the return value from the method invocation by the execution environment is used in application termination (§8.2).

Other than the situations listed above, entry point methods behave like those that are not entry points in every respect. In particular, if the entry point is invoked at any other point during the application's lifetime, such as by regular method invocation, there is no special handling of the method: if there is a parameter, it may have an initial value of null, or a non-null value referring to an array that contains null references. Likewise, the return value of the entry point has no special significance other than in the invocation from the execution environment.

# 8.2 Application termination

Application termination returns control to the execution environment.

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

If the return type of the entry point method is void, reaching the right brace (}) that terminates that method, or executing a return statement that has no expression, results in a termination status code of 0. If the entry point method terminates due to an exception (§21.4), the exit code is implementation-specific. Additionally, the implementation may provide alternative APIs for specifying the exit code.

Prior to an application's termination, an implementation should make every reasonable effort to call finalizers (§15.13) for all of its objects that have not yet been garbage collected, unless such cleanup has been suppressed (by a call to the library method GC.SuppressFinalize, for example). The implementation should document any conditions under which this behavior cannot be guaranteed.

# 8.3 Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces. These are introduced using namespace declarations (§14), which can contain type declarations and nested namespace declarations. Type declarations (§14.7) are used to define classes (§15), structs (§16), interfaces (§18), enums (§19), and delegates (§20). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants (§15.4), fields (§15.5), methods (§15.6), properties (§15.7), events (§15.8), indexers (§15.9), operators (§15.10), instance constructors (§15.11), static constructors (§15.12), finalizers (§15.13), and nested types (§15.3.9).

A declaration defines a name in the *declaration space* to which the declaration belongs. It is a compiletime error to have two or more declarations that introduce members with the same name in a declaration space, except in the following cases:

- Two or more namespace declarations with the same name are allowed in the same declaration space. Such namespace declarations are aggregated to form a single logical namespace and share a single declaration space.
- Declarations in separate programs but in the same namespace declaration space are allowed to share the same name. [*Note*: However, these declarations could introduce ambiguities if included in the same application. *end note*]
- Two or more methods with the same name but distinct signatures are allowed in the same declaration space (§8.6).
- Two or more type declarations with the same name but distinct numbers of type parameters are allowed in the same declaration space (§8.8.2).

- Two or more type declarations with the partial modifier in the same declaration space may share the same name, same number of type parameters and same classification (class, struct or interface). In this case, the type declarations contribute to a single type and are themselves aggregated to form a single declaration space (§15.2.7).
- A namespace declaration and a type declaration in the same declaration space can share the same name as long as the type declaration has at least one type parameter (§8.8.2).

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

- Within all source files of a program, *namespace-member-declarations* with no enclosing *namespace-declaration* are members of a single combined declaration space called the *global declaration space*.
- Within all source files of a program, *namespace-member-declarations* within *namespace-declarations* that have the same fully qualified namespace name are members of a single combined declaration space.
- Each *compilation-unit* and *namespace-body* has an *alias declaration space*. Each *extern-alias-directive* and *using-alias-directive* of the *compilation-unit* or *namespace-body* contributes a member to the alias declaration space (§14.5.2).
- Each non-partial class, struct, or interface declaration creates a new declaration space. Each partial class, struct, or interface declaration contributes to a declaration space shared by all matching parts in the same program (§16.2.3).Names are introduced into this declaration space through *class-member-declarations, struct-member-declarations, interface-member-declarations, or type-parameters*. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded instance constructors and operators. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations differ in their signature (§8.6). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus, a derived class or interface is allowed to declare a member with the same name as an inherited member.
- Each delegate declaration creates a new declaration space. Names are introduced into this declaration space through formal parameters (*fixed-parameters* and *parameter-arrays*) and *type-parameters*.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum-member-declarations*.
- Each method declaration, property declaration, property accessor declaration, indexer declaration, indexer accessor declaration, operator declaration, instance constructor declaration and anonymous function creates a new declaration space called a *local variable declaration space*. Names are introduced into this declaration space through formal parameters (*fixed-parameters* and *parameter-arrays*) and *type-parameters*. The set accessor for a property or an indexer introduces the valuename as a formal parameter. The body of the function member or anonymous function, if any, is considered to be nested within the local variable declaration space to contain elements with the same name. Thus, within a nested local variable declaration space to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.
- Each *block* or *switch-block*, as well as a for, foreach, and using statement, creates a local variable declaration space for local variables and local constants. Names are introduced into this declaration space through *local-variable-declarations* and *local-constant-declarations*. Note that blocks that occur as or within the body of a function member or anonymous function are nested within the local variable declaration space declared by those functions for their parameters. Thus, it is an error to have, for example, a method with a local variable and a parameter of the same name.

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

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

- Declaration order for field declarations determines the order in which their initializers (if any) are executed (§15.5.6.2, §15.5.6.3).
- Local variables shall be defined before they are used (§8.7).
- Declaration order for enum member declarations (§19.4) is significant when *constant-expression* values are omitted.

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

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
    namespace Megacorp.Data
{
        class Order
        {
        ...
    }
}
```

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

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

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }
}
```

```
void H() {
    if (true) {
        int i = 0;
    }
    if (true) {
        int i = 1;
    }
}
void I() {
    for (int i = 0; i < 10; i++)
        H();
    for (int i = 0; i < 10; i++)
        H();
    }
}</pre>
```

end note]

# 8.4 Members

# 8.4.1 General

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

Members of a type are either declared in the type declaration or *inherited* from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, finalizers, and static constructors become members of the derived type. The declared accessibility of a base class member does not control whether the member is inherited—inheritance extends to any member that isn't an instance constructor, static constructor, or finalizer. [*Note*: However, an inherited member might not be accessible in a derived type, for example because of its declared accessibility (§8.5.2). *end note*]

## 8.4.2 Namespace members

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

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

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

## 8.4.3 Struct members

The members of a struct are the members declared in the struct and the members inherited from the struct's direct base class System.ValueType and the indirect base class object.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type (§9.3.5).

## 8.4.4 Enumeration members

The members of an enumeration are the constants declared in the enumeration and the members inherited from the enumeration's direct base class System.Enum and the indirect base classes System.ValueType and object.

## 8.4.5 Class members

The members of a class are the members declared in the class and the members inherited from the base class (except for class object which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not

the instance constructors, finalizers, and static constructors of the base class. Base class members are inherited without regard to their accessibility.

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

The members of object (§9.2.3) and string (§9.2.5) correspond directly to the members of the class types they alias.

# 8.4.6 Interface members

The members of an interface are the members declared in the interface and in all base interfaces of the interface. [*Note*: The members in class object are not, strictly speaking, members of any interface (§18.4). However, the members in class object are available via member lookup in any interface type (§12.5). *end note*]

# 8.4.7 Array members

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

# 8.4.8 Delegate members

A delegate inherits members from class System.Delegate. Additionally, it contains a method named Invoke with the same return type and formal parameter list specified in its declaration (§20.2). An invocation of this method shall behave identically to a delegate invocation (§20.6) on the same delegate instance.

An implementation may provide additional members, either through inheritance or directly in the delegate itself.

# 8.5 Member access

## 8.5.1 General

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (§8.5.2) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be *accessible*. Conversely, when access to a particular member is disallowed, the member is said to be *inaccessible*. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain (§8.5.3) of the member.

# 8.5.2 Declared accessibility

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

- Public, which is selected by including a public modifier in the member declaration. The intuitive meaning of public is "access not limited".
- Protected, which is selected by including a protected modifier in the member declaration. The intuitive meaning of protected is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an internal modifier in the member declaration. The intuitive meaning of internal is "access limited to this assembly".
- Protected internal, which is selected by including both a protected and an internal modifier in the member declaration. The intuitive meaning of protected internal is "accessible within this assembly as well as types derived from the containing class".
- Private, which is selected by including a private modifier in the member declaration. The intuitive meaning of private is "access limited to the containing type".

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

- Namespaces implicitly have public declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared directly in compilation units or namespaces (as opposed to within other types) can have public or internal declared accessibility and default to internal declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to private declared accessibility. [*Note*: A type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. *end note*]
- Struct members can have public, internal, or private declared accessibility and default to
  private declared accessibility because structs are implicitly sealed. Struct members introduced in a
  struct (that is, not inherited by that struct) cannot have protected or protected internal
  declared accessibility. [Note: A type declared as a member of a struct can have public, internal, or
  private declared accessibility, whereas a type declared as a member of a namespace can have only
  public or internal declared accessibility. end note]
- Interface members implicitly have public declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have public declared accessibility. No access modifiers are allowed on enumeration member declarations.

# 8.5.3 Accessibility domains

The *accessibility domain* of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be *top-level* if it is not declared within a type, and a member is said to be *nested* if it is declared within another type. Furthermore, the *program text* of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained in the *type-declarations* of that type (including, possibly, types that are nested within the type).

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

The accessibility domain of a top-level unbound type T (§9.4.4) that is declared in a program P is defined as follows:

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

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

The accessibility domain for a constructed type  $T < A_1, ..., A_N >$  is the intersection of the accessibility domain of the unbound generic type T and the accessibility domains of the type arguments  $A_1, ..., A_N$ .

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

- If the declared accessibility of M is public, the accessibility domain of M is the accessibility domain of T.
- If the declared accessibility of M is protected internal, let D be the union of the program text of P and the program text of any type derived from T, which is declared outside P. The accessibility domain of M is the intersection of the accessibility domain of T with D.

- If the declared accessibility of M is protected, let D be the union of the program text of T and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is internal, the accessibility domain of M is the intersection of the accessibility domain of T with the program text of P.
- If the declared accessibility of M is private, the accessibility domain of M is the program text of T.

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

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

- First, if M is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if M is public, the access is permitted.
- Otherwise, if M is protected internal, the access is permitted if it occurs within the program in which M is declared, or if it occurs within a class derived from the class in which M is declared and takes place through the derived class type (§8.5.4).
- Otherwise, if M is protected, the access is permitted if it occurs within the class in which M is declared, or if it occurs within a class derived from the class in which M is declared and takes place through the derived class type (§8.5.4).
- Otherwise, if M is internal, the access is permitted if it occurs within the program in which M is declared.
- Otherwise, if M is private, the access is permitted if it occurs within the type in which M is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

#### end note]

[Example: In the following code

```
public class A
    public static int X;
    internal static int Y;
private static int Z;
}
internal class B
    public static int X;
    internal static int Y;
private static int Z;
    public class C
        public static int X;
        internal static int Y;
private static int Z;
    }
    private class D
        public static int X;
internal static int Y;
        private static int Z;
    }
}
```

the classes and members have the following accessibility domains:

- The accessibility domain of A and A.X is unlimited.
- The accessibility domain of A.Y, B, B.X, B.Y, B.C, B.C.X, and B.C.Y is the program text of the containing program.
- The accessibility domain of A.Z is the program text of A.
- The accessibility domain of B.Z and B.D is the program text of B, including the program text of B.C and B.D.
- The accessibility domain of B.C.Z is the program text of B.C.
- The accessibility domain of B.D.X and B.D.Y is the program text of B, including the program text of B.C and B.D.
- The accessibility domain of B.D.Z is the program text of B.D.

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

As described in §8.4, all members of a base class, except for instance constructors, finalizers, and static constructors, are inherited by derived types. This includes even private members of a base class. However, the accessibility domain of a private member includes only the program text of the type in which the member is declared. [*Example*: In the following code

```
class A
{
    int x;
    static void F(B b) {
        b.x = 1; // 0k
    }
}
class B: A
{
    static void F(B b) {
        b.x = 1; // Error, x not accessible
    }
}
```

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

## 8.5.4 Protected access

When a protected instance member is accessed outside the program text of the class in which it is declared, and when a protected internal instance member is accessed outside the program text of the program in which it is declared, the access shall take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place *through* an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

Let B be a base class that declares a protected instance member M, and let D be a class that derives from B. Within the *class-body* of D, access to M can take one of the following forms:

- An unqualified type-name or primary-expression of the form M.
- A *primary-expression* of the form E.M, provided the type of E is T or a class derived from T, where T is the class D, or a class type constructed from D.
- A primary-expression of the form base.M.

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a *constructor-initializer* (§15.11.2).

[Example: In the following code

```
public class A
{
    protected int x;
    static void F(A a, B b) {
        a.x = 1; // 0k
        b.x = 1; // 0k
    }
}
public class B: A
{
    static void F(A a, B b) {
        a.x = 1; // Error, must access through instance of B
        b.x = 1; // 0k
    }
}
```

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

[Example:

```
class C<T>
{
    protected T x;
}
class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

Here, the three assignments to x are permitted because they all take place through instances of class types constructed from the generic type. *end example*]

[*Note*: The accessibility domain (§8.5.3) of a protected member declared in a generic class includes the program text of all class declarations derived from any type constructed from that generic class. In the example:

```
class C<T>
{
    protected static T x;
}
class D: C<string>
{
    static void Main() {
        C<int>.x = 5;
    }
}
```

the reference to protected member C<int>.x in D is valid even though the class D derives from C<string>. end note]

# 8.5.5 Accessibility constraints

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

The following accessibility constraints exist:

- The direct base class of a class type shall be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type shall be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type shall be at least as accessible as the delegate type itself.
- The type of a constant shall be at least as accessible as the constant itself.
- The type of a field shall be at least as accessible as the field itself.
- The return type and parameter types of a method shall be at least as accessible as the method itself.
- The type of a property shall be at least as accessible as the property itself.
- The type of an event shall be at least as accessible as the event itself.
- The type and parameter types of an indexer shall be at least as accessible as the indexer itself.
- The return type and parameter types of an operator shall be at least as accessible as the operator itself.
- The parameter types of an instance constructor shall be at least as accessible as the instance constructor itself.

[Example: In the following code

class A {...} public class B: A {...}

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

[Example: Likewise, in the following code

```
class A {...}
public class B
{
    A F() {...}
    internal A G() {...}
    public A H() {...}
}
```

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

# 8.6 Signatures and overloading

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

- The signature of a method consists of the name of the method, the number of type parameters, and the type and parameter-passing mode (value, reference, or output) of each of its formal parameters, considered in the order left to right. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type parameter list of the method. The signature of a method specifically does not include the return type, parameter names, type parameter names, type parameter constraints, the parameter modifiers, nor whether parameters are required or optional.
- The signature of an instance constructor consists of the type and parameter-passing mode (value, reference, or output) of each of its formal parameters, considered in the order left to right. The

signature of an instance constructor specifically does not include the params modifier that may be specified for the right-most parameter.

- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type, nor does it include the params modifier that may be specified for the right-most parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.
- The signature of a conversion operator consists of the source type and the target type. The implicit or explicit classification of a conversion operator is not part of the signature.
- Two signatures of the same member kind (method, instance constructor, indexer or operator) are considered to be the *same signatures* if they have the same name, number of type parameters, number of parameters, and parameter-passing modes, and an identity conversion exists between the types of their corresponding parameters (§11.2.2).

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

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.
- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.
- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

Although out and ref parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by ref and out. A compile-time error occurs if two members are declared in the same type with signatures that would be the same if all parameters in both methods with out modifiers were changed to ref modifiers. For other purposes of signature matching (e.g., hiding or overriding), ref and out are considered part of the signature and do not match each other. [*Note*: This restriction is to allow C# programs to be easily translated to run on the Common Language Infrastructure (CLI), which does not provide a way to define methods that differ solely in ref and out. end note]

The types object and dynamic are not distinguished when comparing signatures. Therefore members declared in a single type whose signatures differ only by replacing object with dynamic are not allowed.

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

interface ITest		
<pre>void F(); void F(int x); void F(ref int x); void F(out int x);</pre>	// F() // F(int) // F(ref int) // F(out int)	error
void F(object o); void F(dynamic d);	// F(object) // error.	
<pre>void F(int x, int y); int F(string s); int F(int x);</pre>	// F(int, int) // F(string) // F(int)	error
void F(string[] a); void F(params string[] a);	// F(string[]) // F(string[])	error
void F <s>(S s); void F<t>(T t);</t></s>	// F<`0>(`0) // F<`0>(`0)	error

Note that any ref and out parameter modifiers (§15.6.2) are part of a signature. Thus, F(int), F(ref int), and F(out int) are all unique signatures. However, F(ref int) and F(out int) cannot be declared within the same interface because their signatures differ solely by ref and out. Also, note that the return type and the params modifier are not part of a signature, so it is not possible to overload solely based on return type or on the inclusion or exclusion of the params modifier. As such, the declarations of the methods F(int) and F(params string[]) identified above, result in a compile-time error. *end example*]

# 8.7 Scopes

# 8.7.1 General

The *scope* of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be *nested*, and an inner scope may redeclare the meaning of a name from an outer scope. (This does not, however, remove the restriction imposed by §8.3 that within a nested block it is not possible to declare a local variable or local constant with the same name as a local variable or local constant in an enclosing block.) The name from the outer scope is then said to be *hidden* in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (§14.6) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is N, is the *namespace-body* of every *namespace-declaration* whose fully qualified name is N or starts with N, followed by a period.
- The scope of a name defined by an *extern-alias-directive* (§14.4) extends over the *using-directives*, *global-attributes* and *namespace-member-declarations* of its immediately containing *compilation-unit* or *namespace-body*. An *extern-alias-directive* does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but, rather, affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a name defined or imported by a *using-directive* (§14.5) extends over the *global-attributes* and *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A *using-directive* may make zero or more namespace or type names available within a particular *compilation-unit* or *namespace-body*, but does not contribute any new members to the underlying declaration space. In other words, a *using-directive* is not transitive but rather affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a type parameter declared by a *type-parameter-list* on a *class-declaration* (§15.2) is the *class-base, type-parameter-constraints-clauses,* and *class-body* of that *class-declaration*. [*Note*: Unlike members of a class, this scope does not extend to derived classes. *end note*]
- The scope of a type parameter declared by a *type-parameter-list* on a *struct-declaration* (§16.2) is the *struct-interfaces*, *type-parameter-constraints-clauses*, and *struct-body* of that *struct-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on an *interface-declaration* (§18.2) is the *interface-base*, *type-parameter-constraints-clauses*, and *interface-body* of that *interface-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on a *delegate-declaration* (§20.2) is the *return-type*, *formal-parameter-list*, and *type-parameter-constraints-clauses* of that *delegate-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on a *method-declaration* (§15.6.1) is the *method-declaration*.

- The scope of a member declared by a *class-member-declaration* (§15.3.1) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (§8.5.3) of the member.
- The scope of a member declared by a *struct-member-declaration* (§16.3) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§19.4) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§15.6) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§15.9) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§15.10) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§15.11) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a parameter declared in a *lambda-expression* (§12.16) is the *lambda-expression-body* of that *lambda-expression*.
- The scope of a parameter declared in an *anonymous-method-expression* (§12.16) is the *block* of that *anonymous-method-expression*.
- The scope of a label declared in a *labeled-statement* (§13.5) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§13.6.2) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *switch-block* of a *switch* statement (§13.8.3) is the *switch-block*.
- The scope of a local variable declared in a *for-initializer* of a for statement (§13.9.4) is the *for-initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the for statement.
- The scope of a local constant declared in a *local-constant-declaration* (§13.6.3) is the *block* in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant-declarator*.
- The scope of a variable declared as part of a *foreach-statement, using-statement, lock-statement* or *query-expression* is determined by the expansion of the given construct.

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

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

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

```
class A
{
    int i = 0;
    void F() {
        i = 1;
        int i;
        i = 2;
    }
```

}

```
void G() {
    int j = (j = 1); // valid
}
void H() {
    int a = 1, b = ++a; // valid
}
```

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

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

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

```
using System;
class A {}
class Test
{
  static void Main() {
    string A = "hello, world";
    string s = A; // expression context
    Type t = typeof(A); // type context
    Console.WriteLine(s); // writes "hello, world"
    Console.WriteLine(t); // writes "A"
}
```

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

# 8.7.2 Name hiding

# 8.7.2.1 General

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

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

# 8.7.2.2 Hiding through nesting

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter, local variable, and local constant declarations. [*Example*: In the following code

;

```
void F() {
    int i = 1;
}
void G() {
    i = 1;
}
```

}

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

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

```
class Outer
{
  static void F(int i) {}
  static void F(string s) {}
  class Inner
  {
    static void F(long l) {}
    void G() {
        F(1); // Invokes Outer.Inner.F
        F("Hello"); // Error
    }
  }
}
```

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

#### 8.7.2.3 Hiding through inheritance

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

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides all non-method base class members with the same name, and all base class methods with the same signature (§8.6).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (§8.6).

The rules governing operator declarations (§15.10) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Contrary to hiding a name from an outer scope, hiding a visible name from an inherited scope causes a warning to be reported. [*Example*: In the following code

```
class Base
{
   public void F() {}
}
class Derived: Base
{
   public void F() {} // Warning, hiding an inherited name
}
```

the declaration of F in Derived causes a warning to be reported. Hiding an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of Base introduced an F method that wasn't present in an earlier version of the class. *end example*]

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

```
class Base
{
   public void F() {}
}
class Derived: Base
{
   new public void F() {}
}
```

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

A declaration of a new member hides an inherited member only within the scope of the new member. [*Example*:

```
class Base
{
    public static void F() {}
}
class Derived: Base
{
    new private static void F() {} // Hides Base.F in Derived only
}
class MoreDerived: Derived
{
    static void G() { F(); } // Invokes Base.F
}
```

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

## 8.8 Namespace and type names

#### 8.8.1 General

Several contexts in a C# program require a namespace-name or a type-name to be specified.

```
namespace-name:

namespace-or-type-name

type-name:

namespace-or-type-name

namespace-or-type-name:

identifier type-argument-list<sub>opt</sub>

namespace-or-type-name . identifier type-argument-list<sub>opt</sub>
```

qualified-alias-member

A namespace-name is a namespace-or-type-name that refers to a namespace.

Following resolution as described below, the *namespace-or-type-name* of a *namespace-name* shall refer to a namespace, or otherwise a compile-time error occurs. No type arguments (§9.4.2) can be present in a *namespace-name* (only types can have type arguments).

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

If the *namespace-or-type-name* is a *qualified-alias-member* its meaning is as described in §14.8.1. Otherwise, a *namespace-or-type-name* has one of four forms:

- I
- I<A1, ..., AK>
- N.I
- N.I<A1, ..., AK>

where I is a single identifier, N is a *namespace-or-type-name* and  $<A_1, ..., A_K>$  is an optional *type-argument-list*. When no *type-argument-list* is specified, consider K to be zero.

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

- If the namespace-or-type-name is a qualified-alias-member, the meaning is as specified in §14.8.1.
- Otherwise, if the *namespace-or-type-name* is of the form I or of the form  $I < A_1, ..., A_K >$ :
  - If K is zero and the *namespace-or-type-name* appears within a generic method declaration (§15.6) but outside the *attributes* of its *method-header*, and if that declaration includes a type parameter (§15.2.3) with name I, then the *namespace-or-type-name* refers to that type parameter.
  - Otherwise, if the *namespace-or-type-name* appears within a type declaration, then for each instance type T (§15.3.2), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
    - If K is zero and the declaration of T includes a type parameter with name I, then the *namespace-or-type-name* refers to that type parameter.
    - Otherwise, if the *namespace-or-type-name* appears within the body of the type declaration, and T or any of its base types contain a nested accessible type having name I and K type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. [*Note:* Non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, finalizers, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace-or-type-name. end note*]
  - Otherwise, for each namespace N, starting with the namespace in which the *namespace-or-type-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
    - If K is zero and I is the name of a namespace in N, then:
      - If the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for N and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name I with a namespace or type, then the *namespace-or-type-name* is ambiguous and a compile-time error occurs.
      - Otherwise, the *namespace-or-type-name* refers to the namespace named I in N.
    - Otherwise, if N contains an accessible type having name I and K type parameters, then:
      - If K is zero and the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for N and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name I with a namespace or type, then the *namespace-or-type-name* is ambiguous and a compile-time error occurs.
      - Otherwise, the *namespace-or-type-name* refers to the type constructed with the given type arguments.
    - Otherwise, if the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for N:

- If K is zero and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name I with an imported namespace or type, then the *namespace-or-type-name* refers to that namespace or type.
- Otherwise, if the namespaces imported by the *using-namespace-directives* of the namespace declaration contain exactly one type having name I and K type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
- Otherwise, if the namespaces imported by the *using-namespace-directives* of the namespace declaration contain more than one type having name I and K type parameters, then the *namespace-or-type-name* is ambiguous and an error occurs.
- Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.
- Otherwise, the *namespace-or-type-name* is of the form N.I or of the form N.I<A<sub>1</sub>, ..., A<sub>K</sub>>. N is first resolved as a *namespace-or-type-name*. If the resolution of N is not successful, a compile-time error occurs. Otherwise, N.I or N.I<A<sub>1</sub>, ..., A<sub>K</sub>> is resolved as follows:
  - If K is zero and N refers to a namespace and N contains a nested namespace with name I, then the *namespace-or-type-name* refers to that nested namespace.
  - Otherwise, if N refers to a namespace and N contains an accessible type having name I and K type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if N refers to a (possibly constructed) class or struct type and N or any of its base classes contain a nested accessible type having name I and K type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. [*Note*: If the meaning of N.I is being determined as part of resolving the base class specification of N then the direct base class of N is considered to be object (§15.2.4.2). *end note*]
  - Otherwise, N. I is an invalid *namespace-or-type-name*, and a compile-time error occurs.

A namespace-or-type-name is permitted to reference a static class (§15.2.2.4) only if

- The namespace-or-type-name is the T in a namespace-or-type-name of the form T.I, or
- The *namespace-or-type-name* is the T in a *typeof-expression* (§12.7.12) of the form typeof(T)

## 8.8.2 Unqualified names

Every namespace declaration and type declaration has an *unqualified name* determined as follows:

- For a namespace declaration, the unqualified name is the *qualified-identifier* specified in the declaration.
- For a type declaration with no *type-parameter-list*, the unqualified name is the *identifier* specified in the declaration.
- For a type declaration with K type parameters, the unqualified name is the *identifier* specified in the declaration, followed by the *generic-dimension-specifier* (§12.7.12) for K type parameters.

## 8.8.3 Fully qualified names

Every namespace and type declaration has a *fully qualified name*, which uniquely identifies the namespace or type declaration amongst all others within the program. The fully qualified name of a namespace or type declaration with unqualified name N is determined as follows:

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

In other words, the fully qualified name of N is the complete hierarchical path of identifiers and *generic-dimension-specifiers* that lead to N, starting from the global namespace. Because every member of a namespace or type shall have a unique name, it follows that the fully qualified name of a namespace or type declaration is always unique. It is a compile-time error for the same fully qualified name to refer to two distinct entities. In particular:

- It is an error for both a namespace declaration and a type declaration to have the same fully qualified name.
- It is an error for two different kinds of type declarations to have the same fully qualified name (for example, if both a struct and class declaration have the same fully qualified name).
- It is an error for a type declaration without the partial modifier to have the same fully qualified name as another type declaration (§15.2.7).

[*Example*: The example below shows several namespace and type declarations along with their associated fully qualified names.

// A
// X
// X.B
// X.B.C
// X.Y
// X.Y.D
// X.Y
// X.Y.E
// X.Y.G<> // X.Y.G<>.H
// X.Y.G<,> // X.Y.G<,>.H<>

end example]

# 8.9 Automatic memory management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a garbage collector. The memory management life cycle of an object is as follows:

- 1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered *live*.
- 2. If neither the object nor any of its instance fields can be accessed by any possible continuation of execution, other than the running of finalizers, the object is considered *no longer in use* and it becomes eligible for finalization. [*Note*: The C# compiler and the garbage collector might choose to analyze code to determine which references to an object might be used in the future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector might (but is not required to) treat the object as no longer in use. *end note*]

- 3. Once the object is eligible for finalization, at some unspecified later time the finalizer (§15.13) (if any) for the object is run. Under normal circumstances the finalizer for the object is run once only, though implementation-specific APIs may allow this behavior to be overridden.
- 4. Once the finalizer for an object is run, if neither the object nor any of its instance fields can be accessed by any possible continuation of execution, including the running of finalizers, the object is considered *inaccessible* and the object becomes eligible for collection. [*Note*: An object which could previously not be accessed may become accessible again due to its finalizer. An example of this is provided below. *end note*]
- 5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage, and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector might implement a wide range of memory management policies. C# requires that finalizers be run at some time between the time an object is eligible and the time that the application exits, but specifies neither a time constraint within that span, nor an order in which finalizers are run.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class System.GC. This class can be used to request a collection to occur, finalizers to be run (or not run), and so forth.

[*Example*: Since the garbage collector is allowed wide latitude in deciding when to collect objects and run finalizers, a conforming implementation might produce output that differs from that shown by the following code. The program

```
using System;
class A
{
   ~A() {
      Console.WriteLine("Finalize instance of A");
   }
}
class B
   object Ref;
   public B(object o) {
    Ref = o;
   }
   ~B() {
      Console.WriteLine("Finalize instance of B");
   }
}
class Test
   static void Main() {
      B b = new B(new A());
      b = null;
      GC.Collect():
      GC.WaitForPendingFinalizers();
   }
}
```

creates an instance of class A and an instance of class B. These objects become eligible for garbage collection when the variable b is assigned the value null, since after this time it is impossible for any user-written code to access them. The output could be either

```
Finalize instance of A
Finalize instance of B
Finalize instance of B
Finalize instance of A
```

or

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

In subtle cases, the distinction between "eligible for finalization" and "eligible for collection" can be important. For example,

```
using System;
class A
{
   ~A() {
       Console.WriteLine("Finalize instance of A");
   }
   public void F() {
      Console.WriteLine("A.F");
      Test.RefA = this;
   }
}
class B
   public A Ref;
   ~B() {
       Console.WriteLine("Finalize instance of B");
      Ref.F();
   }
}
class Test
   public static A RefA;
   public static B RefB;
   static void Main() {
      RefB = new B();
      RefA = new A()
       RefB.Ref = RefA;
      RefB = null;
      RefA = null:
       // A and B now eligible for finalization
      GC.Collect();
      GC.WaitForPendingFinalizers();
// B now eligible for collection, but A is not
if (RefA != null)
          Console.WriteLine("RefA is not null");
   }
}
```

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

```
Finalize instance of A
Finalize instance of B
A.F
RefA is not null
```

Note that although the instance of A was not in use and A's finalizer was run, it is still possible for methods of A (in this case, F) to be called from another finalizer. Also, note that running of a finalizer might cause an object to become usable from the mainline program again. In this case, the running of B's finalizer caused an instance of A that was previously not in use, to become accessible from the live reference Test.RefA. After the call to WaitForPendingFinalizers, the instance of B is eligible for collection, but the instance of A is not, because of the reference Test.RefA. *end example*]

# 8.10 Execution order

Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A *side effect* is defined as a read or write of a volatile field, a write to a non-volatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects shall be preserved are references to volatile fields (§15.5.4), lock statements (§13.13), and thread creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints:

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved (§15.5.5, §15.5.6).
- The ordering of side effects is preserved with respect to volatile reads and writes (§15.5.4). Additionally, the execution environment need not evaluate part of an expression if it can deduce that that expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects are visible in the original program order.

# 9. Types

# 9.1 General

The types of the C# language are divided into two main categories: *reference types* and *value types*. Both value types and reference types may be *generic types*, which take one or more *type parameters*. Type parameters can designate both value types and reference types.

type: reference-type value-type type-parameter

A third category of types, pointers, is available only in unsafe code (§23.3).

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store *references* to their data, the latter being known as *objects*. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. [*Note*: When a variable is a ref or out parameter, it does not have its own storage but references the storage of another variable. In this case, the ref or out variable is effectively an alias for another variable and not a distinct variable. *end note*]

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the object class type, and object is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type object. Values of value types are treated as objects by performing boxing and unboxing operations (§9.3.12).

# 9.2 Reference types

# 9.2.1 General

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

reference-type: class-type interface-type array-type delegate-type dynamic class-type: type-name object string interface-type: type-name array-type: non-array-type rank-specifiers

```
non-array-type:
    value-type
    class-type
    interface-type
    delegate-type
    dynamic
    type-parameter
rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier
rank-specifier:
    [ dim-separators<sub>opt</sub> ]
dim-separators:
    dim-separators ,
delegate-type:
    type-name
```

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

#### 9.2.2 Class types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, finalizers, and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using *object-creation-expressions* (§12.7.11.2).

Class types are described in §15.

Certain predefined class types have special meaning in the C# language, as described in the table below.

Class type	Description
System.Object	The ultimate base class of all other types. See §9.2.3.
System.String	The string type of the C# language. See §9.2.5.
System.ValueType	The base class of all value types. See 9.3.2.
System.Enum	The base class of all enum types. See §19.5.
System.Array	The base class of all array types. See §17.2.2.
System.Delegate	The base class of all delegate types. See §20.1.
System.Exception	The base class of all exception types. See §21.3.

## 9.2.3 The object type

The object class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the object class type.

The keyword object is simply an alias for the predefined class System.Object.

#### 9.2.4 The dynamic type

The dynamic type, like object, can reference any object. When operations are applied to expressions of type dynamic, their resolution is deferred until the program is run. Thus, if the operation cannot

legitimately be applied to the referenced object, no error is given during compilation. Instead, an exception will be thrown when resolution of the operation fails at run-time.

The dynamic type is further described in §9.7, and dynamic binding in §12.3.1.

## 9.2.5 The string type

The string type is a sealed class type that inherits directly from object. Instances of the string class represent Unicode character strings.

Values of the string type can be written as string literals (§7.4.5.6).

The keyword string is simply an alias for the predefined class System.String.

#### 9.2.6 Interface types

An interface defines a contract. A class or struct that implements an interface shall adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interface types are described in §18.

#### 9.2.7 Array types

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

Array types are described in §17.

#### 9.2.8 Delegate types

A delegate is a data structure that refers to one or more methods. For instance methods, it also refers to their corresponding object instances.

[*Note*: The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance on which to invoke the method. *end note*]

Delegate types are described in §20.

## 9.3 Value types

## 9.3.1 General

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

```
value-type:
struct-type
enum-type
struct-type:
type-name
simple-type
nullable-value-type
simple-type:
numeric-type
```

bool

```
numeric-type:
   integral-type
   floating-point-type
    decimal
integral-type:
    sbyte
    byte
    short
    ushort
    int
    uint
    long
    ulong
    char
floating-point-type:
    float
    double
nullable-type:
    non-nullable-value-type ?
non-nullable-value-type:
    type
enum-type:
    type-name
```

Unlike a variable of a reference type, a variable of a value type can contain the value null only if the value type is a nullable value type (§9.3.11). For every non-nullable value type there is a corresponding nullable value type denoting the same set of values plus the value null.

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

# 9.3.2 The System.ValueType type

All value types implicitly inherit from the class System.ValueType, which, in turn, inherits from class object. It is not possible for any type to derive from a value type, and value types are thus implicitly sealed (§15.2.2.3).

Note that System.ValueType is not itself a *value-type*. Rather, it is a *class-type* from which all *value-types* are automatically derived.

# 9.3.3 Default constructors

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

- For all *simple-types*, the default value is the value produced by a bit pattern of all zeros:
  - For sbyte, byte, short, ushort, int, uint, long, and ulong, the default value is 0.
  - $\circ$  For char, the default value is '\x0000'.
  - For float, the default value is 0.0f.
  - For double, the default value is 0.0d.
  - For decimal, the default value is 0.0m.
  - For bool, the default value is false.
- For an *enum-type* E, the default value is 0, converted to the type E.

- For a *struct-type*, the default value is the value produced by setting all value type fields to their default value and all reference type fields to null.
- For a *nullable-value-type* the default value is an instance for which the HasValue property is false. The default value is also known as the *null value* of the nullable value type. Attempting to read the Value property of such a value causes an exception of type System.InvalidOperationException to be thrown (§9.3.11).

Like any other instance constructor, the default constructor of a value type is invoked using the new operator. [*Note*: For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. For value types, the default value expression (§12.7.15) produces the same result as using the default constructor. *end note*] [*Example*: In the code below, variables i, j and k are all initialized to zero.

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
        int k = default(int);
    }
}
```

## end example]

Because every value type implicitly has a public parameterless instance constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized instance constructors (§16.4.9).

## 9.3.4 Struct types

A struct type is a value type that can declare constants, fields, methods, properties, events, indexers, operators, instance constructors, static constructors, and nested types. The declaration of struct types is described in §16.

# 9.3.5 Simple types

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

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

Because a simple type aliases a struct type, every simple type has members. [*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
```

*end example*] [*Note*: 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* (§7.4.5). [*Example*: 123 is a literal of type int and 'a' is a literal of type char. *end example*] C# makes no provision for literals of struct types in general.
- 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* (§12.20). 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 (§15.4). It is not possible to have constants of other struct types, but a similar effect is provided by static readonly fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined conversionoperator (§11.5.3). *end note*]

# 9.3.6 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 from -128 to 127, inclusive.
- The byte type represents unsigned 8-bit integers with values from 0 to 255, inclusive.
- The short type represents signed 16-bit integers with values from -32768 to 32767, inclusive.
- The ushort type represents unsigned 16-bit integers with values from 0 to 65535, inclusive.
- The int type represents signed 32-bit integers with values from -2147483648 to 2147483647, inclusive.
- The uint type represents unsigned 32-bit integers with values from 0 to 4294967295, inclusive.
- The long type represents signed 64-bit integers with values from -9223372036854775808 to 9223372036854775807, inclusive.
- The ulong type represents unsigned 64-bit integers with values from 0 to 18446744073709551615, inclusive.
- The char type represents unsigned 16-bit integers with values from 0 to 65535, inclusive. The set of possible values for the char type corresponds to the Unicode character set. [*Note*: Although char has the same representation as ushort, not all operations permitted on one type are permitted on the other. *end note*]

The *integral-type* unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision, as detailed in §12.4.7.

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

- There are no predefined implicit conversions from other types to the char type. In particular, even though the byte and ushort types have ranges of values that are fully representable using the char type, implicit conversions from sbyte, byte, or ushort to char do not exist.
- Constants of the char type shall be written as *character-literals* or as *integer-literals* in combination with a cast to type char. [*Example*: (char)10 is the same as '\x000A'. *end example*]

The checked and unchecked operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions (§12.7.14). In a checked context, an overflow produces a compile-time error or causes a System.OverflowException to be thrown. In an unchecked context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

# 9.3.7 Floating-point types

C# supports two floating-point types: float and double. The float and double types are represented using the 32-bit single-precision and 64-bit double-precision IEC 60559 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (§12.9.3).
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. [*Example*: 1.0 / 0.0 yields positive infinity, and -1.0 / 0.0 yields negative infinity. *end example*]
- The *Not-a-Number* value, often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form s × m × 2<sup>e</sup>, where s is 1 or −1, and m and e are determined by the particular floating-point type: For float, 0 < m < 2<sup>24</sup> and −149 ≤ e ≤ 104, and for double, 0 < m < 2<sup>53</sup> and −1075 ≤ e ≤ 970. Denormalized floating-point numbers are considered valid non-zero values. C# neither requires nor forbids that a conforming implementation support denormalized floating-point numbers.

The float type can represent values ranging from approximately  $1.5 \times 10^{-45}$  to  $3.4 \times 10^{38}$  with a precision of 7 digits.

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

If either operand of a binary operator is a floating-point type then standard numeric promotions are applied, as detailed in §12.4.7, and the operation is performed with float or double precision.

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

- The result of a floating-point operation is rounded to the nearest representable value in the destination format.
- If the magnitude of the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the magnitude of the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

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

infinity. To force a value of a floating-point type to the exact precision of its type, an explicit cast can be used. *end example*]

# 9.3.8 The decimal type

The decimal type is a 128-bit data type suitable for financial and monetary calculations. The decimal type can represent values including those in the range at least  $-7.9 \times 10^{-28}$  to  $7.9 \times 10^{28}$ , with at least 28-digit precision.

The finite set of values of type decimal are of the form  $(-1)^s \times c \times 10^{-e}$ , where the sign s is 0 or 1, the coefficient c is given by  $0 \le c < Cmax$ , and the scale e is such that  $Emin \le e \le Emax$ , where Cmax is at least  $1 \times 10^{28}$ ,  $Emin \le 0$ , and  $Emax \ge 28$ . The decimal type does not necessarily support signed zeros, infinities, or NaN's.

A decimal is represented as an integer scaled by a power of ten. For decimals with an absolute value less than 1.0m, the value is exact to at least the 28<sup>th</sup> decimal place. For decimals with an absolute value greater than or equal to 1.0m, the value is exact to at least 28 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 often have non-terminating binary expansions, making those representations more prone to round-off errors.

If either operand of a binary operator is of decimal type then standard numeric promotions are applied, as detailed in §12.4.7, and the operation is performed with double precision.

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 at least the 28<sup>th</sup> decimal place. Note that rounding may produce a zero value from a non-zero value.

If a decimal arithmetic operation produces a result whose magnitude is too large for the decimal format, a System.OverflowException is thrown.

The decimal type has greater precision but may have a 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 or overflow exceptions. For these reasons, no implicit conversions exist between the floating-point types and decimal, and without explicit casts, a compile-time error occurs when floating-point and decimal operands are directly mixed in the same expression.

# 9.3.9 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 value types. In particular, the bool type is distinct and separate from the integral types, a bool value cannot be used in place of an integral value, and vice versa.

[*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*]

# 9.3.10 Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which shall be byte, sbyte, short, ushort, int, uint, long or ulong. The set of values of the

enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations (§19.2).

# 9.3.11 Nullable value types

A nullable value type can represent all values of its *underlying type* plus an additional null value. A nullable value type is written T?, where T is the underlying type. This syntax is shorthand for System.Nullable<T>, and the two forms can be used interchangeably.

Conversely, a *non-nullable value type* is any value type other than System.Nullable<T> and its shorthand T? (for any T), plus any type parameter that is constrained to be a non-nullable value type (that is, any type parameter with a value type constraint (§15.2.5)). The System.Nullable<T> type specifies the value type constraint for T, which means that the underlying type of a nullable value type can be any non-nullable value type. The underlying type of a nullable value type or a reference type. For example, int?? and string? are invalid types.

An instance of a nullable value type T? has two public read-only properties:

- A HasValue property of type bool
- A Value property of type T

An instance for which HasValue is true is said to be non-null. A non-null instance contains a known value and Value returns that value.

An instance for which HasValue is false is said to be null. A null instance has an undefined value. Attempting to read the Value of a null instance causes a System.InvalidOperationException to be thrown. The process of accessing the Value property of a nullable instance is referred to as **unwrapping**.

In addition to the default constructor, every nullable value type T? has a public constructor with a single parameter of type T. Given a value x of type T, a constructor invocation of the form

new T?(x)

creates a non-null instance of T? for which the Value property is x. The process of creating a non-null instance of a nullable value type for a given value is referred to as *wrapping*.

Implicit conversions are available from the null literal to T? (§11.2.6) and from T to T? (§11.2.5).

The nullable type T? implements no interfaces (§18). In particular, this means it does not implement any interface that the underlying type T does.

# 9.3.12 Boxing and unboxing

The concept of boxing and unboxing provide a bridge between *value-types* and *reference-types* by permitting any value of a *value-type* to be converted to and from type object. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

Boxing is described in more detail in §11.2.8 and unboxing is described in §11.3.6.

# 9.4 Constructed types

# 9.4.1 General

A generic type declaration, by itself, denotes an *unbound generic type* that is used as a "blueprint" to form many different types, by way of applying *type arguments*. The type arguments are written within angle brackets (< and >) immediately following the name of the generic type. A type that includes at least one type argument is called a *constructed type*. A constructed type can be used in most places in the language in which a type name can appear. An unbound generic type can only be used within a *typeof-expression* (§12.7.12).

Constructed types can also be used in expressions as simple names (§12.7.3) or when accessing a member (§12.7.5).

When a *namespace-or-type-name* is evaluated, only generic types with the correct number of type parameters are considered. Thus, it is possible to use the same identifier to identify different types, as long as the types have different numbers of type parameters. This is useful when mixing generic and non-generic classes in the same program. [*Example*:

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}
namespace MyApplication
{
    using Widgets;
    class X
    {
        Queue q1; // Non-generic Widgets.Queue
        Queue<int> q2; // Generic Widgets.Queue
    }
}
```

```
end example]
```

The detailed rules for name lookup in the *namespace-or-type-name* productions is described in §8.8. The resolution of ambiguities in these productions is described in §7.2.5. A *type-name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup (§15.3.9.7). [*Example*:

```
class Outer<T>
{
    public class Inner {...}
    public Inner i; // Type of i is Outer<T>.Inner
}
```

end example]

A non-enum constructed type shall not be used as an *unmanaged-type* (§23.3).

# 9.4.2 Type arguments

Each argument in a type argument list is simply a type.

```
type-argument-list:
    < type-arguments >
type-arguments:
    type-argument
    type-arguments , type-argument
type-argument:
    type
```

A *type-argument* shall not be a pointer type (§23). Each type argument shall satisfy any constraints on the corresponding type parameter (§15.2.5).

## 9.4.3 Open and closed types

All types can be classified as either **open types** or **closed types**. An open type is a type that involves type parameters. More specifically:

• A type parameter defines an open type.

- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments is an open type. A constructed nested type is an open type if and only if one or more of its type arguments or the type arguments of its containing type(s) is an open type.

A closed type is a type that is not an open type.

At run-time, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic type is bound to a particular run-time type. The run-time processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open type does not exist at run-time, there are no static variables associated with an open type. Two closed constructed types are the same type if they are constructed from the same unbound generic type, and their corresponding type arguments are the same type.

# 9.4.4 Bound and unbound types

The term *unbound type* refers to a non-generic type or an unbound generic type. The term *bound type* refers to a non-generic type or a constructed type.

An unbound type refers to the entity declared by a type declaration. An unbound generic type is not itself a type, and cannot be used as the type of a variable, argument or return value, or as a base type. The only construct in which an unbound generic type can be referenced is the typeof expression (§12.7.12).

# 9.4.5 Satisfying constraints

Whenever a constructed type or generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or method (§15.2.5). For each where clause, the type argument A that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a class type, an interface type, or a type parameter, let C represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it shall be the case that type A is convertible to type C by one of the following:
  - An identity conversion (§11.2.2)
  - An implicit reference conversion (§11.2.7)
  - A boxing conversion (§11.2.8), provided that type A is a non-nullable value type.
  - $\circ$  An implicit reference, boxing or type parameter conversion from a type parameter A to C.
- If the constraint is the reference type constraint (class), the type A shall satisfy one of the following:
  - A is an interface type, class type, delegate type, array type or the dynamic type. [Note:
     System.ValueType and System.Enum are reference types that satisfy this constraint. end note]
  - $\circ~$  A is a type parameter that is known to be a reference type (§9.2).
- If the constraint is the value type constraint (struct), the type A shall satisfy one of the following:
  - A is a struct type or enum type, but not a nullable value type. [Note: System.ValueType and System.Enum are reference types that do not satisfy this constraint. end note]
  - A is a type parameter having the value type constraint (§15.2.5).
- If the constraint is the constructor constraint new(), the type A shall not be abstract and shall have a public parameterless constructor. This is satisfied if one of the following is true:
  - A is a value type, since all value types have a public default constructor (§9.3.3).
  - $\circ~$  A is a type parameter having the constructor constraint (§15.2.5).

- A is a type parameter having the value type constraint (§15.2.5).
- A is a class that is not abstract and contains an explicitly declared public constructor with no parameters.
- A is not abstract and has a default constructor (§15.11.5).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

Since type parameters are not inherited, constraints are never inherited either. [*Example*: In the following, D needs to specify the constraint on its type parameter T so that T satisfies the constraint imposed by the base class B<T>. In contrast, class E need not specify a constraint, because List<T> implements IEnumerable for any T.

```
class B<T> where T: IEnumerable {...}
class D<T>: B<T> where T: IEnumerable {...}
class E<T>: B<List<T>> {...}
```

end example]

# 9.5 Type parameters

A type parameter is an identifier designating a value type or reference type that the parameter is bound to at run-time.

type-parameter: identifier

Since a type parameter can be instantiated with many different type arguments, type parameters have slightly different operations and restrictions than other types. [*Note*: These include:

- A type parameter cannot be used directly to declare a base class (§15.2.4.2) or interface (§18.2.4).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in §12.5.
- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in §11.2.11 and §11.3.8.
- The literal null cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type (§11.2.11). However, a default expression (§12.7.15) can be used instead. In addition, a value with a type given by a type parameter *can* be compared with null using == and != (§12.11.7) unless the type parameter has the value type constraint.
- A new expression (§12.7.11.2) can only be used with a type parameter if the type parameter is constrained by a *constructor-constraint* or the value type constraint (§15.2.5).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access (§12.7.5) or type name (§8.8) to identify a static member or a nested type.
- A type parameter cannot be used as an unmanaged-type (§23.3).

#### end note]

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed constructed type (§9.4.3). The run-time execution of all statements and expressions involving type parameters uses the type that was supplied as the type argument for that parameter.

## 9.6 Expression tree types

*Expression trees* permit lambda expressions to be represented as data structures instead of executable code. Expression trees are values of *expression tree types* of the form

System.Linq.Expressions.Expression<TDelegate>, where TDelegate is any delegate type. For the remainder of this specification we will refer to these types using the shorthand Expression<TDelegate>.

If a conversion exists from a lambda expression to a delegate type D, a conversion also exists to the expression tree type Expression<TDelegate>. Whereas the conversion of a lambda expression to a delegate type generates a delegate that references executable code for the lambda expression, conversion to an expression tree type creates an expression tree representation of the lambda expression.

Expression trees are efficient in-memory data representations of lambda expressions and make the structure of the lambda expression transparent and explicit.

Just like a delegate type D, Expression<TDelegate> is said to have parameter and return types, which are the same as those of D.

[*Example*: The following program represents a lambda expression both as executable code and as an expression tree. Because a conversion exists to Func<int,int>, a conversion also exists to Expression<Func<int,int>>:

Func <int, int=""> del = <math>x \Rightarrow x + 1</math>;</int,>	// Code
<pre>Expression<func<int,int>&gt; exp = x =&gt; x + 1;</func<int,int></pre>	// Data

Following these assignments, the delegate del references a method that returns x + 1, and the expression tree exp references a data structure that describes the expression  $x \Rightarrow x + 1$ . *end example*]

The exact definition of the generic type Expression<TDelegate> as well as the precise rules for constructing an expression tree when a lambda expression is converted to an expression tree type, are implementation dependent.

Two things are important to make explicit:

- Not all lambda expressions can be converted to expression trees. For instance, lambda expressions with statement bodies, and lambda expressions containing assignment expressions cannot be represented. In these cases, a conversion still exists, but it will fail at compile-time. These exceptions are detailed in §11.7.3.
- Expression<TDelegate> offers an instance method Compile which produces a delegate of type TDelegate:

Func<int,int> del2 = exp.Compile();

Invoking this delegate causes the code represented by the expression tree to be executed. Thus, given the definitions above, del and del2 are equivalent, and the following two statements will have the same effect:

int i1 = del(1); int i2 = del2(1);

After executing this code, i1 and i2 will both have the value 2.

# 9.7 The dynamic type

The type dynamic has special meaning in C#. Its purpose is to allow dynamic binding, which is described in detail in §12.3.2.

dynamic is considered identical to object except in the following respects:

- Operations on expressions of type dynamic can be dynamically bound (§12.3.3).
- Type inference (§12.6.3) will prefer dynamic over object if both are candidates.
- dynamic cannot be used as
  - the type in an *object-creation-expression* (§12.7.11.2)
  - a predefined-type in a member-access (§12.7.5.1)

- the operand of the typeof operator
- o an attribute argument
- o a constraint
- $\circ$  an extension method type
- o any part of a type argument within *struct-interfaces* (§16.2.4) or *interface-type-list* (§15.2.4.1).

Because of this equivalence, the following holds:

- There is an implicit identity conversion between object and dynamic, and between constructed types that are the same when replacing dynamic with object
- Implicit and explicit conversions to and from object also apply to and from dynamic.
- Signatures that are the same when replacing dynamic with object are considered the same signature
- The type dynamic is indistinguishable from object at run-time.
- An expression of the type dynamic is referred to as a *dynamic expression*.

# **10. Variables**

# 10.1 General

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment or through use of the ++ and -- operators.

A variable shall be *definitely assigned* (§10.4) before its value can be obtained.

As described in the following subclauses, variables are either *initially assigned* or *initially unassigned*. An initially assigned variable has a well-defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable shall occur in every possible execution path leading to that location.

# **10.2 Variable categories**

## 10.2.1 General

C# defines seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The subclauses that follow describe each of these categories.

[Example: In the following code

```
class A
{
    public static int x;
    int y;
    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
            c = a + b++;
    }
}
```

x is a static variable, y is an instance variable, v[0] is an array element, a is a value parameter, b is a reference parameter, c is an output parameter, and i is a local variable. *end example*]

## 10.2.2 Static variables

A field declared with the static modifier is called a *static variable*. A static variable comes into existence before execution of the static constructor (§15.12) for its containing type, and ceases to exist when the associated application domain ceases to exist.

The initial value of a static variable is the default value (§10.3) of the variable's type.

For the purposes of definite assignment checking, a static variable is considered initially assigned.

## 10.2.3 Instance variables

10.2.3.1 General

A field declared without the static modifier is called an *instance variable*.

## 10.2.3.2 Instance variables in classes

An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's finalizer (if any) has executed.

The initial value of an instance variable of a class is the default value (§10.3) of the variable's type.

For the purpose of definite assignment checking, an instance variable of a class is considered initially assigned.

## 10.2.3.3 Instance variables in structs

An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance variables of the struct.

The initial assignment state of an instance variable of a struct is the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables are likewise unassigned.

# 10.2.4 Array elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

The initial value of each of the elements of an array is the default value (§10.3) of the type of the array elements.

For the purpose of definite assignment checking, an array element is considered initially assigned.

## 10.2.5 Value parameters

A parameter declared without a ref or out modifier is a value parameter.

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) or anonymous function to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter normally ceases to exist when execution of the function body completes. However, if the value parameter is captured by an anonymous function (§12.16.6.2), its lifetime extends at least until the delegate or expression tree created from that anonymous function is eligible for garbage collection.

For the purpose of definite assignment checking, a value parameter is considered initially assigned.

# 10.2.6 Reference parameters

A parameter declared with a ref modifier is a *reference parameter*.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member or anonymous function invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters. [*Note*: The rules for output parameters are different, and are described in §10.2.7. *end note*]

- A variable shall be definitely assigned (§10.4) before it can be passed as a reference parameter in a function member or delegate invocation.
- Within a function member or anonymous function, a reference parameter is considered initially assigned.

For a struct type, within an instance method or instance accessor (§12.2.1) or instance constructor with a constructor initializer, the this keyword behaves exactly as a reference parameter of the struct type (§12.7.8).

## 10.2.7 Output parameters

A parameter declared with an out modifier is an output parameter.

An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member or delegate invocation. Thus, the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. [*Note*: The rules for reference parameters are different, and are described in §10.2.6. *end note*]

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member or delegate invocation.
- Following the normal completion of a function member or delegate invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member or anonymous function, an output parameter is considered initially unassigned.
- Every output parameter of a function member or anonymous function shall be definitely assigned (§10.4) before the function member or anonymous function returns normally.

Within an instance constructor of a struct type, the this keyword behaves exactly as an output or reference parameter of the struct type, depending on whether the constructor declaration includes a constructor initializer (§12.7.8).

## 10.2.8 Local variables

A **local variable** is declared by a *local-variable-declaration*, *foreach-statement*, or *specific-catch-clause* of a *try-statement*. For a *foreach-statement*, the local variable is an iteration variable (§13.9.5). For a *specific-catch-clause*, the local variable is an exception variable (§13.11). A local variable declared by a *foreach-statement* or *specific-catch-clause* is considered initially assigned.

#### A local-variable-declaration can occur in a block, a for-statement, a switch-block, or a using-statement.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends from entry into the scope with which it is associated, at least until execution of that scope ends in some way. (Entering an enclosed *block*, calling a method, or yielding a value from an iterator block suspends, but does not end, execution of the current scope.) If the local variable is captured by an anonymous function (§12.16.6.2), its lifetime extends at least until the delegate or expression tree created from the anonymous function, along with any other objects that come to reference the captured variable, are eligible for garbage collection. If the parent scope is entered recursively or iteratively, a new instance of the local variable is instantiated each time, and its *local-variable-initializer*, if any, is evaluated each time. [*Note*: A local variable is instantiated each time its scope is entered. This behavior is visible to user code containing anonymous methods. *end note*] [*Note*: The lifetime of an *iteration variable* (§13.9.5) declared by a *foreach-statement* is a single iteration of that statement. Each iteration creates a new variable. *end note*] [*Note*: The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is only used for a small portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable (§8.9). *end note*]

A local variable introduced by a *local-variable-declaration* is not automatically initialized and thus has no default value. Such a local variable is considered initially unassigned. [*Note*: A *local-variable-declaration*]

that includes a *local-variable-initializer* is still initially unassigned. Execution of the declaration behaves exactly like an assignment to the variable (§10.4.4.5). It is possible to use a variable without executing its *local-variable-initializer*; e.g., within the initializer expression itself or by using a *goto-statement* to bypass the initialization:

```
goto L;
int x = 1; // never executed
L: x += 1; // error: x not definitely assigned
```

end note]

Within the scope of a local variable, it is a compile-time error to refer to that local variable in a textual position that precedes its *local-variable-declarator*.

# 10.3 Default values

The following categories of variables are automatically initialized to their default values:

- Static variables.
- Instance variables of class instances.
- Array elements.

The default value of a variable depends on the type of the variable and is determined as follows:

- For a variable of a *value-type*, the default value is the same as the value computed by the *value-type*'s default constructor (§9.3.3).
- For a variable of a *reference-type*, the default value is null.

[*Note*: Initialization to default values is typically done by having the memory manager or garbage collector initialize memory to all-bits-zero before it is allocated for use. For this reason, it is convenient to use all-bits-zero to represent the null reference. *end note*]

# 10.4 Definite assignment

## 10.4.1 General

At a given location in the executable code of a function member or an anonymous function, a variable is said to be *definitely assigned* if the compiler can prove, by a particular static flow analysis (§10.4.4), that the variable has been automatically initialized or has been the target of at least one assignment. [*Note*: Informally stated, the rules of definite assignment are:

- An initially assigned variable (§10.4.2) is always considered definitely assigned.
- An initially unassigned variable (§10.4.3) is considered definitely assigned at a given location if all possible execution paths leading to that location contain at least one of the following:
  - A simple assignment (§12.18.2) in which the variable is the left operand.
  - An invocation expression (§12.7.6) or object creation expression (§12.7.11.2) that passes the variable as an output parameter.
  - For a local variable, a local variable declaration for the variable (§13.6.2) that includes a variable initializer.

The formal specification underlying the above informal rules is described in §10.4.2, §10.4.3, and §10.4.4. *end note*]

The definite assignment states of instance variables of a *struct-type* variable are tracked individually as well as collectively. In additional to the rules above, the following rules apply to *struct-type* variables and their instance variables:

• An instance variable is considered definitely assigned if its containing *struct-type* variable is considered definitely assigned.

• A *struct-type* variable is considered definitely assigned if each of its instance variables is considered definitely assigned.

Definite assignment is a requirement in the following contexts:

- A variable shall be definitely assigned at each location where its value is obtained. [*Note*: This ensures that undefined values never occur. *end note*] The occurrence of a variable in an expression is considered to obtain the value of the variable, except when
  - o the variable is the left operand of a simple assignment,
  - o the variable is passed as an output parameter, or
  - the variable is a *struct-type* variable and occurs as the left operand of a member access.
- A variable shall be definitely assigned at each location where it is passed as a reference parameter. [*Note*: This ensures that the function member being invoked can consider the reference parameter initially assigned. *end note*]
- All output parameters of a function member shall be definitely assigned at each location where the function member returns (through a return statement or through execution reaching the end of the function member body). [*Note*: This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable. *end note*]
- The this variable of a *struct-type* instance constructor shall be definitely assigned at each location where that instance constructor returns.

#### 10.4.2 Initially assigned variables

The following categories of variables are classified as initially assigned:

- Static variables.
- Instance variables of class instances.
- Instance variables of initially assigned struct variables.
- Array elements.
- Value parameters.
- Reference parameters.
- Variables declared in a catch clause or a foreach statement.

## 10.4.3 Initially unassigned variables

The following categories of variables are classified as initially unassigned:

- Instance variables of initially unassigned struct variables.
- Output parameters, including the this variable of struct instance constructors without a constructor initializer.
- Local variables, except those declared in a catch clause or a foreach statement.

#### 10.4.4 Precise rules for determining definite assignment

#### 10.4.4.1 General

In order to determine that each used variable is definitely assigned, the compiler shall use a process that is equivalent to the one described in this subclause.

The compiler processes the body of each function member that has one or more initially unassigned variables. For each initially unassigned variable *v*, the compiler determines a *definite assignment state* for *v* at each of the following points in the function member:

- At the beginning of each statement
- At the end point (§13.2) of each statement
- On each arc which transfers control to another statement or to the end point of a statement

- At the beginning of each expression
- At the end of each expression

The definite assignment state of v can be either:

- Definitely assigned. This indicates that on all possible control flows to this point, *v* has been assigned a value.
- Not definitely assigned. For the state of a variable at the end of an expression of type bool, the state of a variable that isn't definitely assigned might (but doesn't necessarily) fall into one of the following sub-states:
  - Definitely assigned after true expression. This state indicates that v is definitely assigned if the Boolean expression evaluated as true, but is not necessarily assigned if the Boolean expression evaluated as false.
  - Definitely assigned after false expression. This state indicates that *v* is definitely assigned if the Boolean expression evaluated as false, but is not necessarily assigned if the Boolean expression evaluated as true.

The following rules govern how the state of a variable v is determined at each location.

## 10.4.4.2 General rules for statements

- *v* is not definitely assigned at the beginning of a function member body.
- The definite assignment state of v at the beginning of any other statement is determined by checking the definite assignment state of v on all control flow transfers that target the beginning of that statement. If (and only if) v is definitely assigned on all such control flow transfers, then v is definitely assigned at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (§13.2).
- The definite assignment state of v at the end point of a block, checked, unchecked, if, while, do, for, foreach, lock, using, or switch statement is determined by checking the definite assignment state of v on all control flow transfers that target the end point of that statement. If v is definitely assigned on all such control flow transfers, then v is definitely assigned at the end point of the statement. Otherwise, v is not definitely assigned at the end point of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (§13.2).

[*Note*: Because there are no control paths to an unreachable statement, *v* is definitely assigned at the beginning of any unreachable statement. *end note*]

## 10.4.4.3 Block statements, checked, and unchecked statements

The definite assignment state of v on the control transfer to the first statement of the statement list in the block (or to the end point of the block, if the statement list is empty) is the same as the definite assignment statement of v before the block, checked, or unchecked statement.

## 10.4.4.4 Expression statements

For an expression statement *stmt* that consists of the expression *expr*:

- v has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* if definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*; otherwise, it is not definitely assigned at the end point of *stmt*.

## 10.4.4.5 Declaration statements

- If *stmt* is a declaration statement without initializers, then *v* has the same definite assignment state at the end point of *stmt* as at the beginning of *stmt*.
- If *stmt* is a declaration statement with initializers, then the definite assignment state for *v* is determined as if *stmt* were a statement list, with one assignment statement for each declaration with an initializer (in the order of declaration).

## 10.4.4.6 If statements

For an if statement *stmt* of the form:

if ( expr ) then-stmt else else-stmt

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If v is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *then-stmt* and to either *else-stmt* or to the end-point of *stmt* if there is no else clause.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *then-stmt*, and not definitely assigned on the control flow transfer to either *else-stmt* or to the end-point of *stmt* if there is no else clause.
- If v has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *else-stmt*, and not definitely assigned on the control flow transfer to *then-stmt*. It is definitely assigned at the end-point of *stmt* if and only if it is definitely assigned at the end-point of *then-stmt*.
- Otherwise, v is considered not definitely assigned on the control flow transfer to either the *then-stmt* or *else-stmt*, or to the end-point of *stmt* if there is no else clause.

#### 10.4.4.7 Switch statements

In a switch statement *stmt* with a controlling expression *expr*:

- The definite assignment state of v at the beginning of *expr* is the same as the state of v at the beginning of *stmt*.
- The definite assignment state of v on the control flow transfer to a reachable switch block statement list is the same as the definite assignment state of v at the end of *expr*.

#### 10.4.4.8 While statements

For a while statement *stmt* of the form:

while ( expr ) while-body

- v has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If v is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *while-body* and to the end point of *stmt*.
- If v has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *while-body*, but not definitely assigned at the end-point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*, but not definitely assigned on the control flow transfer to *while-body*.

## 10.4.4.9 Do statements

For a do statement *stmt* of the form:

```
do do-body while ( expr ) ;
```

- v has the same definite assignment state on the control flow transfer from the beginning of *stmt* to *dobody* as at the beginning of *stmt*.
- *v* has the same definite assignment state at the beginning of *expr* as at the end point of *do-body*.
- If v is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.
- If v has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*, but not definitely assigned on the control flow transfer to *do-body*.

#### 10.4.4.10 For statements

Definite assignment checking for a for statement of the form:

```
for ( for-initializer ; for-condition ; for-iterator ) embedded-statement
```

is done as if the statement were written:

```
{
    for-initializer ;
    while ( for-condition ) {
        embedded-statement ;
    LLoop: for-iterator ;
    }
}
```

with continue statements that target the for statement being translated to goto statements targeting the label LLoop. If the *for-condition* is omitted from the for statement, then evaluation of definite assignment proceeds as if *for-condition* were replaced with true in the above expansion.

10.4.4.11 Break, continue, and goto statements

The definite assignment state of v on the control flow transfer caused by a break, continue, or goto statement is the same as the definite assignment state of v at the beginning of the statement.

#### 10.4.4.12 Throw statements

For a statement stmt of the form

throw *expr* ;

the definite assignment state of v at the beginning of *expr* is the same as the definite assignment state of v at the beginning of *stmt*.

#### 10.4.4.13 Return statements

For a statement stmt of the form

return expr ;

- The definite assignment state of v at the beginning of *expr* is the same as the definite assignment state of v at the beginning of *stmt*.
- If v is an output parameter, then it shall be definitely assigned either:
  - o after expr
  - or at the end of the finally block of a try-finally or try-catch-finally that encloses the return statement.

For a statement *stmt* of the form:

return ;

- If v is an output parameter, then it shall be definitely assigned either:
  - o before *stmt*
  - or at the end of the finally block of a try-finally or try-catch-finally that encloses the return statement.

10.4.4.14 Try-catch statements

For a statement *stmt* of the form:

try try-block
catch ( ... ) catch-block-1
...
catch ( ... ) catch-block-n

- The definite assignment state of v at the beginning of *try-block* is the same as the definite assignment state of v at the beginning of *stmt*.
- The definite assignment state of v at the beginning of *catch-block-i* (for any *i*) is the same as the definite assignment state of v at the beginning of *stmt*.
- The definite assignment state of v at the end-point of *stmt* is definitely assigned if (and only if) v is definitely assigned at the end-point of *try-block* and every *catch-block-i* (for every *i* from 1 to *n*).

10.4.4.15 Try-finally statements

For a try statement *stmt* of the form:

try try-block finally finally-block

- The definite assignment state of v at the beginning of *try-block* is the same as the definite assignment state of v at the beginning of *stmt*.
- The definite assignment state of v at the beginning of *finally-block* is the same as the definite assignment state of v at the beginning of *stmt*.
- The definite assignment state of v at the end-point of *stmt* is definitely assigned if (and only if) at least one of the following is true:
  - v is definitely assigned at the end-point of try-block
  - v is definitely assigned at the end-point of *finally-block*

If a control flow transfer (such as a goto statement) is made that begins within *try-block*, and ends outside of *try-block*, then *v* is also considered definitely assigned on that control flow transfer if *v* is definitely assigned at the end-point of *finally-block*. (This is not an only if—if *v* is definitely assigned for another reason on this control flow transfer, then it is still considered definitely assigned.)

10.4.4.16 Try-catch-finally statements

Definite assignment analysis for a try-catch-finally statement of the form:

```
try try-block
catch ( ... ) catch-block-1
...
catch ( ... ) catch-block-n
finally finally-block
```

is done as if the statement were a try-finally statement enclosing a try-catch statement:

```
try {
    try try-block
    catch ( ... ) catch-block-1
    ...
    catch ( ... ) catch-block-n
}
finally finally-block
```

[*Example*: The following example demonstrates how the different blocks of a try statement (§13.11) affect definite assignment.

```
class A
{
  static void F() {
    int i, j;
    try {
      goto LABEL;
      // neither i nor j definitely assigned
      i = 1;
      // i definitely assigned
    }
```

```
catch {
    // neither i nor j definitely assigned
    i = 3;
    // i definitely assigned
}
finally {
    // neither i nor j definitely assigned
    j = 5;
    // j definitely assigned
}
// i and j definitely assigned
LABEL:;
// j definitely assigned
}
```

end example]

}

10.4.4.17 Foreach statements

For a foreach statement *stmt* of the form:

foreach ( type identifier in expr ) embedded-statement

- The definite assignment state of v at the beginning of *expr* is the same as the state of v at the beginning of *stmt*.
- The definite assignment state of v on the control flow transfer to *embedded-statement* or to the end point of *stmt* is the same as the state of v at the end of *expr*.

# 10.4.4.18 Using statements

For a using statement *stmt* of the form:

```
using ( resource-acquisition ) embedded-statement
```

- The definite assignment state of v at the beginning of *resource-acquisition* is the same as the state of v at the beginning of *stmt*.
- The definite assignment state of v on the control flow transfer to *embedded-statement* is the same as the state of v at the end of *resource-acquisition*.

10.4.4.19 Lock statements

For a lock statement *stmt* of the form:

lock ( expr ) embedded-statement

- The definite assignment state of v at the beginning of *expr* is the same as the state of v at the beginning of *stmt*.
- The definite assignment state of v on the control flow transfer to *embedded-statement* is the same as the state of v at the end of *expr*.

## 10.4.4.20 Yield statements

For a yield return statement *stmt* of the form:

yield return expr ;

- The definite assignment state of v at the beginning of *expr* is the same as the state of v at the beginning of *stmt*.
- The definite assignment state of v at the end of *stmt* is the same as the state of v at the end of *expr*.

A yield break statement has no effect on the definite assignment state.

10.4.4.21 General rules for constant expressions

The following applies to any constant expression, and takes priority over any rules from the following sections that might apply:

For a constant expression with value true:

- If v is definitely assigned before the expression, then v is definitely assigned after the expression.
- Otherwise v is "definitely assigned after false expression" after the expression.

[Example:

```
int x;
if (true) {}
else
{
Console.WriteLine(x);
}
```

end example]

For a constant expression with value false:

- If v is definitely assigned before the expression, then v is definitely assigned after the expression.
- Otherwise v is "definitely assigned after true expression" after the expression.

[Example:

```
int x;
if (false)
{
   Console.WriteLine(x);
}
```

end example]

For all other constant expressions, the definite assignment state of v after the expression is the same as the definite assignment state of v before the expression.

# 10.4.4.22 General rules for simple expressions

The following rule applies to these kinds of expressions: literals (§12.7.2), simple names (§12.7.3), member access expressions (§12.7.5), non-indexed base access expressions (§12.7.9), typeof expressions (§12.7.12), and default value expressions (§12.7.15).

• The definite assignment state of v at the end of such an expression is the same as the definite assignment state of v at the beginning of the expression.

# 10.4.4.23 General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions (§12.7.4), element access expressions (§12.7.7), base access expressions with indexing (§12.7.9), increment and decrement expressions (§12.7.10, §12.8.6), cast expressions (§12.8.7), unary +, -, ~, \* expressions, binary +, -, \*, /, %, <<<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ expressions (§12.9, §12.10, §12.11, §12.12), compound assignment expressions (§12.7.11), and await expressions (§12.8.8).

Each of these expressions has one or more subexpressions that are unconditionally evaluated in a fixed order. [*Example*: The binary % operator evaluates the left hand side of the operator, then the right hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. *end example*] For an expression *expr*, which has subexpressions *expr*<sub>1</sub>, *expr*<sub>2</sub>, ..., *expr*<sub>n</sub>, evaluated in that order:

• The definite assignment state of v at the beginning of *expr*<sub>1</sub> is the same as the definite assignment state at the beginning of *expr*.

- The definite assignment state of v at the beginning of *expr<sub>i</sub>* (*i* greater than one) is the same as the definite assignment state at the end of *expr<sub>i-1</sub>*.
- The definite assignment state of v at the end of *expr* is the same as the definite assignment state at the end of *expr<sub>n</sub>*.

## 10.4.4.24 Invocation expressions and object creation expressions

If the method to be invoked is a partial method that has no implementing partial method declaration, or is a conditional method for which the call is omitted ( $\S$ 22.5.3.2), then the definite assignment state of *v* after the invocation is the same as the definite assignment state of *v* before the invocation. Otherwise the following rules apply:

For an invocation expression *expr* of the form:

primary-expression (  $arg_1$ ,  $arg_2$ , ... ,  $arg_n$  )

or an object creation expression expr of the form:

new type (  $arg_1$ ,  $arg_2$ , ... ,  $arg_n$  )

- For an invocation expression, the definite assignment state of v before *primary-expression* is the same as the state of v before *expr*.
- For an invocation expression, the definite assignment state of v before arg<sub>1</sub> is the same as the state of v after primary-expression.
- For an object creation expression, the definite assignment state of v before arg<sub>1</sub> is the same as the state of v before expr.
- For each argument *arg<sub>i</sub>*, the definite assignment state of *v* after *arg<sub>i</sub>* is determined by the normal expression rules, ignoring any ref or out modifiers.
- For each argument *arg<sub>i</sub>* for any *i* greater than one, the definite assignment state of *v* before *arg<sub>i</sub>* is the same as the state of *v* after *arg<sub>i-1</sub>*.
- If the variable v is passed as an out argument (i.e., an argument of the form "out v") in any of the arguments, then the state of v after *expr* is definitely assigned. Otherwise, the state of v after *expr* is the same as the state of v after *arg<sub>n</sub>*.
- For array initializers (§12.7.11.5), object initializers (12.7.11.3), collection initializers (§12.7.11.4) and anonymous object initializers (§12.7.11.7), the definite assignment state is determined by the expansion that these constructs are defined in terms of.

10.4.4.25 Simple assignment expressions

For an expression *expr* of the form w = expr-rhs:

- The definite assignment state of v before w is the same as the definite assignment state of v before *expr*.
- The definite assignment state of v before *expr-rhs* is the same as the definite assignment state of v after w.
- If *w* is the same variable as *v*, then the definite assignment state of *v* after *expr* is definitely assigned. Otherwise, the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr*.

[Example: In the following code

```
class A
{
    static void F(int[] arr) {
        int x;
        arr[x = 1] = x; // ok
    }
}
```

the variable x is considered definitely assigned after arr[x = 1] is evaluated as the left hand side of the second simple assignment. *end example*]

#### 10.4.4.26 && expressions

For an expression *expr* of the form *expr-first* && *expr-second*:

- The definite assignment state of *v* before *expr-first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of v before *expr-second* is definitely assigned if and only if the state of v after *expr-first* is either definitely assigned or "definitely assigned after true expression". Otherwise, it is not definitely assigned.
- The definite assignment state of *v* after *expr* is determined by:
  - If the state of *v* after *expr-first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr-second* is definitely assigned, and the state of *v* after *expr-first* is "definitely assigned after false expression", then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr-second* is definitely assigned or "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after true expression".
  - Otherwise, if the state of *v* after *expr-first* is "definitely assigned after false expression", and the state of *v* after *expr-second* is "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after false expression".
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

[Example: In the following code

```
class A
{
   static void F(int x, int y) {
      int i;
      if (x >= 0 && (i = y) >= 0) {
        // i definitely assigned
      }
      else {
        // i not definitely assigned
      }
      // i not definitely assigned
   }
}
```

the variable i is considered definitely assigned in one of the embedded statements of an if statement but not in the other. In the if statement in method F, the variable i is definitely assigned in the first embedded statement because execution of the expression (i = y) always precedes execution of this embedded statement. In contrast, the variable i is not definitely assigned in the second embedded statement, since x >= 0 might have tested false, resulting in the variable i's being unassigned. *end example*]

## 10.4.4.27 || expressions

For an expression *expr* of the form *expr-first* || *expr-second*:

- The definite assignment state of v before *expr-first* is the same as the definite assignment state of v before *expr*.
- The definite assignment state of v before *expr-second* is definitely assigned if and only if the state of v after *expr-first* is either definitely assigned or "definitely assigned after true expression". Otherwise, it is not definitely assigned.
- The definite assignment statement of *v* after *expr* is determined by:

- If the state of *v* after *expr-first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
- Otherwise, if the state of *v* after *expr-second* is definitely assigned, and the state of *v* after *expr-first* is "definitely assigned after true expression", then the state of *v* after *expr* is definitely assigned.
- Otherwise, if the state of *v* after *expr-second* is definitely assigned or "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after false expression".
- Otherwise, if the state of v after *expr-first* is "definitely assigned after true expression", and the state of v after *expr-second* is "definitely assigned after true expression", then the state of v after *expr* is "definitely assigned after true expression".
- Otherwise, the state of *v* after *expr* is not definitely assigned.

[Example: In the following code

```
class A
{
  static void G(int x, int y) {
    int i;
    if (x >= 0 || (i = y) >= 0) {
        // i not definitely assigned
     }
     else {
        // i definitely assigned
     }
     // i not definitely assigned
   }
}
```

the variable i is considered definitely assigned in one of the embedded statements of an if statement but not in the other. In the if statement in method G, the variable i is definitely assigned in the second embedded statement because execution of the expression (i = y) always precedes execution of this embedded statement. In contrast, the variable i is not definitely assigned in the first embedded statement, since x >= 0 might have tested true, resulting in the variable i's being unassigned. *end example*]

# 10.4.4.28 ! expressions

For an expression *expr* of the form ! *expr-operand*:

- The definite assignment state of *v* before *expr-operand* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
  - If the state of v after *expr-operand* is definitely assigned, then the state of v after *expr* is definitely assigned.
  - Otherwise, if the state of v after *expr-operand* is "definitely assigned after false expression", then the state of v after *expr* is "definitely assigned after true expression".
  - Otherwise, if the state of v after *expr-operand* is "definitely assigned after true expression", then the state of v after *expr* is "definitely assigned after false expression".
  - Otherwise, the state of v after *expr* is not definitely assigned.

# 10.4.4.29 ?? expressions

For an expression *expr* of the form *expr-first* ?? *expr-second*:

- The definite assignment state of v before *expr-first* is the same as the definite assignment state of v before *expr*.
- The definite assignment state of v before *expr-second* is the same as the definite assignment state of v after *expr-first*.
- The definite assignment statement of *v* after *expr* is determined by:

- If *expr-first* is a constant expression (§12.20) with value null, then the state of *v* after *expr* is the same as the state of *v* after *expr-second*.
- Otherwise, the state of *v* after *expr* is the same as the definite assignment state of *v* after *expr-first*.

#### 10.4.4.30 ?: expressions

For an expression *expr* of the form *expr-cond* ? *expr-true* : *expr-false*:

- The definite assignment state of v before *expr-cond* is the same as the state of v before *expr*.
- The definite assignment state of v before *expr-true* is definitely assigned if the state of v after *expr-cond* is definitely assigned or "definitely assigned after true expression".
- The definite assignment state of v before *expr-false* is definitely assigned if the state of v after *expr-cond* is definitely assigned or "definitely assigned after false expression".
- The definite assignment state of *v* after *expr* is determined by:
  - If *expr-cond* is a constant expression (§12.20) with value true then the state of *v* after *expr* is the same as the state of *v* after *expr-true*.
  - Otherwise, if *expr-cond* is a constant expression (§12.20) with value false then the state of *v* after *expr* is the same as the state of *v* after *expr-false*.
  - Otherwise, if the state of *v* after *expr-true* is definitely assigned and the state of *v* after *expr-false* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

#### 10.4.4.31 Anonymous functions

For a *lambda-expression* or *anonymous-method-expression expr* with a body (either *block* or *expression*) *body*:

- The definite assignment state of a parameter is the same as for a parameter of a named method (§10.2.6, §10.2.7).
- The definite assignment state of an outer variable *v* before *body* is the same as the state of *v* before *expr*. That is, definite assignment state of outer variables is inherited from the context of the anonymous function.
- The definite assignment state of an outer variable *v* after *expr* is the same as the state of *v* before *expr*.

[Example: The example

```
delegate bool Filter(int i);
void F() {
    int max;
    // Error, max is not definitely assigned
    Filter f = (int n) => n < max;
    max = 5;
    Dowork(f);
}
```

generates a compile-time error since max is not definitely assigned where the anonymous function is declared. *end example*] [*Example*: The example

```
delegate void D();
void F() {
    int n;
    D d = () => { n = 1; };
    d();
    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

also generates a compile-time error since the assignment to n in the anonymous function has no affect on the definite assignment state of n outside the anonymous function. *end example*]

# **10.5 Variable references**

A *variable-reference* is an *expression* that is classified as a variable. A *variable-reference* denotes a storage location that can be accessed both to fetch the current value and to store a new value.

variable-reference: expression

[Note: In C and C++, a variable-reference is known as an lvalue. end note]

# 10.6 Atomicity of variable references

Reads and writes of the following data types shall be atomic: bool, char, byte, sbyte, short, ushort, uint, int, float, and reference types. In addition, reads and writes of enum types with an underlying type in the previous list shall also be atomic. Reads and writes of other types, including long, ulong, double, and decimal, as well as user-defined types, need not be atomic. Aside from the library functions designed for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment or decrement.

# 11. Conversions

# 11.1 General

A *conversion* causes an expression to be converted to, or treated as being of, a particular type; in the former case a conversion may involve a change in representation. Conversions can be *implicit* or *explicit*, and this determines whether an explicit cast is required. [*Example*: For instance, the conversion from type int to type long is implicit, so expressions of type int can implicitly be treated as type long. The opposite conversion, from type long to type int, is explicit and so an explicit cast is required.

*end example*] Some conversions are defined by the language. Programs may also define their own conversions (§11.5).

Some conversions in the language are defined from expressions to types, others from types to types. A conversion from a type applies to all expressions that have that type. [*Example*:

```
enum Color { Red, Blue, Green }
```

```
Color c0 = 0; // The expression 0 converts implicitly to enum types
Color c1 = (Color)1; // other int expressions need explicit conversion
String x = null; // Conversion from null expression (no type) to String
Func<int, int> square = x => x * x; // Conversion from lambda expression
to delegate type
```

end example]

## 11.2 Implicit conversions

#### 11.2.1 General

The following conversions are classified as implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit enumeration conversions
- Implicit reference conversions
- Boxing conversions
- Implicit dynamic conversions
- Implicit type parameter conversions
- Implicit constant expression conversions
- User-defined implicit conversions
- Anonymous function conversions
- Method group conversions
- Null literal conversions
- Implicit nullable conversions
- Lifted user-defined implicit conversions

Implicit conversions can occur in a variety of situations, including function member invocations (§12.6.6), cast expressions (§12.8.7), and assignments (§12.18).

The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. [*Note*: Properly designed user-defined implicit conversions should exhibit these characteristics as well. *end note*]

For the purposes of conversion, the types object and dynamic are considered equivalent.

However, dynamic conversions (§11.2.9 and §11.3.7) apply only to expressions of type dynamic (§9.2.4).

## 11.2.2 Identity conversion

An identity conversion converts from any type to the same type. One reason this conversion exists is so that a type T or an expression of type T can be said to be convertible to T itself.

Because object and dynamic are considered equivalent there is an identity conversion between object and dynamic, and between constructed types that are the same when replacing all occurrences of dynamic with object.

In most cases, an identity conversion has no effect at runtime. However, since floating point operations may be performed at higher precision than prescribed by their type (§9.3.7), assignment of their results may result in a loss of precision, and explicit casts are guaranteed to reduce precision to what is prescribed by the type.

## 11.2.3 Implicit numeric conversions

The implicit numeric conversions are:

- From sbyte to short, int, long, float, double, or decimal.
- From byte to short, ushort, int, uint, long, ulong, float, double, or decimal.
- From short to int, long, float, double, or decimal.
- From ushort to int, uint, long, ulong, float, double, or decimal.
- From int to long, float, double, or decimal.
- From uint to long, ulong, float, double, or decimal.
- From long to float, double, or decimal.
- From ulong to float, double, or decimal.
- From char to ushort, int, uint, long, ulong, float, double, or decimal.
- From float to double.

Conversions from int, uint, long or ulong to float and from long or ulong to double may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

There are no predefined implicit conversions to the char type, so values of the other integral types do not automatically convert to the char type.

## 11.2.4 Implicit enumeration conversions

An implicit enumeration conversion permits the *decimal-integer-literal* 0 (or 0L, etc.) to be converted to any *enum-type* and to any *nullable-value-type* whose underlying type is an *enum-type*. In the latter case the conversion is evaluated by converting to the underlying *enum-type* and wrapping the result (§9.3.11).

# 11.2.5 Implicit nullable conversions

The implicit nullable conversions are those nullable conversions (§11.6.1) derived from implicit predefined conversions.

# 11.2.6 Null literal conversions

An implicit conversion exists from the null literal to any reference type or nullable value type. This conversion produces a null reference if the target type is a reference type, or the null value (§9.3.11) of the given nullable value type.

#### 11.2.7 Implicit reference conversions

The implicit reference conversions are:

• From any *reference-type* to object and dynamic.

- From any *class-type* S to any *class-type* T, provided S is derived from T.
- From any *class-type* S to any *interface-type* T, provided S implements T.
- From any *interface-type* S to any *interface-type* T, provided S is derived from T.
- From an *array-type* S with an element type S<sub>E</sub> to an *array-type* T with an element type T<sub>E</sub>, provided all of the following are true:
  - S and T differ only in element type. In other words, S and T have the same number of dimensions.
  - $\circ~$  An implicit reference conversion exists from  $S_E$  to  $T_E.$
- From a single-dimensional array type S[] to System.Collections.Generic.IList<T>, System.Collections.Generic.IReadOnlyList<T>, and their base interfaces, provided that there is an implicit identity or reference conversion from S to T.
- From any *array-type* to System. Array and the interfaces it implements.
- From any *delegate-type* to System.Delegate and the interfaces it implements.
- From the null literal (§7.4.5.7) to any reference-type.
- From any *reference-type* to a *reference-type* T if it has an implicit identity or reference conversion to a *reference-type* T<sub>0</sub> and T<sub>0</sub> has an identity conversion to T.
- From any *reference-type* to an interface or delegate type T if it has an implicit identity or reference conversion to an interface or delegate type T<sub>0</sub> and T<sub>0</sub> is variance-convertible (§18.2.3.3) to T.
- Implicit conversions involving type parameters that are known to be reference types. See §11.2.11 for more details on implicit conversions involving type parameters.

The implicit reference conversions are those conversions between *reference-types* that can be proven to always succeed, and therefore require no checks at run-time.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. [*Note*: In other words, while a reference conversion can change the type of the reference, it never changes the type or value of the object being referred to. *end note*]

# 11.2.8 Boxing conversions

A boxing conversion permits a *value-type* to be implicitly converted to a *reference-type*. The following boxing conversions exist:

- From any *value-type* to the type object.
- From any *value-type* to the type System.ValueType.
- From any *enum-type* to the type System. Enum.
- From any *non-nullable-value-type* to any *interface-type* implemented by the *non-nullable-value-type*.
- From any *non-nullable-value-type* to any *interface-type* I such that there is a boxing conversion from the *non-nullable-value-type* to another *interface-type* IO, and IO has an identity conversion to I.
- From any *non-nullable-value-type* to any *interface-type* I such that there is a boxing conversion from the *non-nullable-value-type* to another *interface-type* IO, and IO is variance-convertible (§18.2.3.3) to I.
- From any *nullable-value-type* to any *reference-type* where there is a boxing conversion from the underlying type of the *nullable-value-type* to the *reference-type*.
- From a type parameter that is not known to be a reference type to any type such that the conversion is permitted by §11.2.11.

Boxing a value of a *non-nullable-value-type* consists of allocating an object instance and copying the value into that instance.

Boxing a value of a *nullable-value-type* produces a null reference if it is the null value (HasValue is false), or the result of unwrapping and boxing the underlying value otherwise.

[*Note*: The process of boxing may be imagined in terms of the existence of a boxing class for every value type. For example, consider a struct S implementing an interface I, with a boxing class called S\_Boxing.

```
interface I
{
   void M();
}
struct S : I
{
   public void M() { ... }
}
sealed class S_Boxing : I
{
   s value:
   public S_Boxing(S value) {
      this.value = value;
   public void M() {
      value.M();
}
```

Boxing a value V of type S now consists of executing the expression new  $S_Boxing(v)$  and returning the resulting instance as a value of the target type of the conversion. Thus, the statements

```
S s = new S();
object box = s;
```

can be thought of as similar to:

```
S s = new S();
object box = new S_Boxing(s);
```

The imagined boxing type described above does not actually exist. Instead, a boxed value of type S has the runtime type S, and a runtime type check using the is operator with a value type as the right operand tests whether the left operand is a boxed version of the right operand. For example,

```
int i = 123;
object box = i;
if (box is int) {
    Console.Write("Box contains an int");
}
```

will output the string "Box contains an int" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a *reference-type* to type object, in which the value continues to reference the same instance and simply is regarded as the less derived type object. For example, given the declaration

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

the following statements

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.write(((Point)box).x);
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of p to box causes the value of p to be copied. Had Point been declared a class instead, the value 20 would be output because p and box would reference the same instance. The analogy of a boxing class should not be used as more than a helpful tool for picturing how boxing works conceptually. There are numerous subtle differences between the behavior described by this specification and the behavior that would result from boxing being implemented in precisely this manner. *end note*]

## 11.2.9 Implicit dynamic conversions

An implicit dynamic conversion exists from an expression of type dynamic to any type T. The conversion is dynamically bound (§12.3.3), which means that an implicit conversion will be sought at run-time from the run-time type of the expression to T. If no conversion is found, a run-time exception is thrown.

This implicit conversion seemingly violates the advice in the beginning of §11.2 that an implicit conversion should never cause an exception. However, it is not the conversion itself, but the *finding* of the conversion that causes the exception. The risk of run-time exceptions is inherent in the use of dynamic binding. If dynamic binding of the conversion is not desired, the expression can be first converted to object, and then to the desired type.

[Example: The following illustrates implicit dynamic conversions:

```
object o = "object"
dynamic d = "dynamic";
string s1 = o; // Fails at compile-time - no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i = d; // Compiles but fails at run-time - no conversion exists
```

The assignments to s2 and i both employ implicit dynamic conversions, where the binding of the operations is suspended until run-time. At run-time, implicit conversions are sought from the run-time type of d - string - to the target type. A conversion is found to string but not to int. *end example*]

## 11.2.10 Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A constant-expression (§12.20) of type int can be converted to type sbyte, byte, short, ushort, uint, or ulong, provided the value of the constant-expression is within the range of the destination type.
- A *constant-expression* of type long can be converted to type ulong, provided the value of the *constant-expression* is not negative.

# 11.2.11 Implicit conversions involving type parameters

For a *type-parameter* T that is known to be a reference type (§15.2.5), the following implicit reference conversions (11.2.7) exist:

- From T to its effective base class C, from T to any base class of C, and from T to any interface implemented by C.
- From T to an *interface-type* I in T's effective interface set and from T to any base interface of I.
- From T to a type parameter U provided that T depends on U (§15.2.5). [*Note*: Since T is known to be a reference type, within the scope of T, the run-time type of U will always be a reference type, even if U is not known to be a reference type at compile-time. *end note*]
- From the null literal (§7.4.5.7) to T.

For a *type-parameter* T that is *not* known to be a reference type ( $\S15.2.5$ ), the following conversions involving T are considered to be boxing conversions (11.2.8) at compile-time. At run-time, if T is a value type, the conversion is executed as a boxing conversion. At run-time, if T is a reference type, the conversion is executed as an implicit reference conversion or identity conversion.

• From T to its effective base class C, from T to any base class of C, and from T to any interface implemented by C. [*Note*: C will be one of the types System.Object, System.ValueType, or System.Enum (otherwise T would be known to be a reference type). *end note*]

• From T to an *interface-type* I in T's effective interface set and from T to any base interface of I.

For a *type-parameter* T that is *not* known to be a reference type, there is an implicit conversion from T to a type parameter U provided T depends on U. At run-time, if T is a value type and U is a reference type, the conversion is executed as a boxing conversion. At run-time, if both T and U are value types, then T and U are necessarily the same type and no conversion is performed. At run-time, if T is a reference type, then U is necessarily also a reference type and the conversion is executed as an implicit reference conversion or identity conversion (§15.2.5).

The following further implicit conversions exist for a given type parameter T:

- From T to a reference type S if it has an implicit conversion to a reference type S<sub>0</sub> and S<sub>0</sub> has an identity conversion to S. At run-time, the conversion is executed the same way as the conversion to S<sub>0</sub>.
- From T to an interface type I if it has an implicit conversion to an interface type I<sub>0</sub>, and I<sub>0</sub> is varianceconvertible to I (§18.2.3.3). At run-time, if T is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

In all cases, the rules ensure that a conversion is executed as a boxing conversion if and only if at run-time the conversion is from a value type to a reference type.

## 11.2.12 User-defined implicit conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined implicit conversions are described in §11.5.4.

## 11.2.13 Anonymous function conversions and method group conversions

Anonymous functions and method groups do not have types in and of themselves, but they may be implicitly converted to delegate types. Additionally, some lambda expressions may be implicitly converted to expression tree types. Anonymous function conversions are described in more detail in §11.7 and method group conversions in §11.8.

# 11.3 Explicit conversions

# 11.3.1 General

The following conversions are classified as explicit conversions:

- All implicit conversions
- Explicit numeric conversions
- Explicit enumeration conversions
- Explicit nullable conversions.
- Explicit reference conversions
- Explicit interface conversions
- Unboxing conversions
- Explicit type parameter conversions
- Explicit dynamic conversions

Explicit conversions can occur in cast expressions (§12.8.7).

The set of explicit conversions includes all implicit conversions. [*Note*: This means that redundant cast expressions are allowed. *end note*]

The explicit conversions that are not implicit conversions are conversions that cannot be proven always to succeed, conversions that are known possibly to lose information, and conversions across domains of types sufficiently different to merit explicit notation.

## 11.3.2 Explicit numeric conversions

The explicit numeric conversions are the conversions from a *numeric-type* to another *numeric-type* for which an implicit numeric conversion (§11.2.3) does not already exist:

- From sbyte to byte, ushort, uint, ulong, or char.
- From byte to sbyte or char.
- From short to sbyte, byte, ushort, uint, ulong, or char.
- From ushort to sbyte, byte, short, or char.
- From int to sbyte, byte, short, ushort, uint, ulong, or char.
- From uint to sbyte, byte, short, ushort, int, or char.
- From long to sbyte, byte, short, ushort, int, uint, ulong, or char.
- From ulong to sbyte, byte, short, ushort, int, uint, long, or char.
- From char to sbyte, byte, or short.
- From float to sbyte, byte, short, ushort, int, uint, long, ulong, char, or decimal.
- From double to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or decimal.
- From decimal to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or double.

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any *numeric-type* to any other *numeric-type* using a cast expression (§12.8.7).

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context (§12.7.14) in which the conversion takes place:
  - In a checked context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a System.OverflowException if the value of the source operand is outside the range of the destination type.
  - o In an unchecked context, the conversion always succeeds, and proceeds as follows.
    - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
    - If the source type is smaller than the destination type, then the source value is either signextended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
    - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type
- For a conversion from decimal to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a System.OverflowException is thrown.
- For a conversion from float or double to an integral type, the processing depends on the overflowchecking context (§12.7.14) in which the conversion takes place:
  - In a checked context, the conversion proceeds as follows:
    - If the value of the operand is NaN or infinite, a System.OverflowException is thrown.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.

- Otherwise, a System.OverflowException is thrown.
- o In an unchecked context, the conversion always succeeds, and proceeds as follows.
  - If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.
  - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.
  - Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from double to float, the double value is rounded to the nearest float value. If the double value is too small to represent as a float, the result becomes zero with the same sign as the value. If the magnitude of the double value is too large to represent as a float, the result becomes infinity with the same sign as the value. If the double value is NaN, the result is also NaN.
- For a conversion from float or double to decimal, the source value is converted to decimal representation and rounded to the nearest number if required (§9.3.8).
  - If the source value is too small to represent as a decimal, the result becomes zero, preserving the sign of the original value if decimal supports signed zero values.
  - If the source value's magnitude is too large to represent as a decimal, or that value is infinity, the result is infinity preserving the sign of the original value, if the decimal representation supports infinities; otherwise a System.OverflowException is thrown.
  - If the source value is NaN, the result is NaN if the decimal representation supports NaNs; otherwise a System.OverflowException is thrown.
- For a conversion from decimal to float or double, the decimal value is rounded to the nearest double or float value. If the source value's magnitude is too large to represent in the target type, or that value is infinity, the result is infinity preserving the sign of the original value. If the source value is NaN, the result is NaN. While this conversion may lose precision, it never causes an exception to be thrown.

[Note: The decimal type is not required to support infinities or NaN values but may do so; its range may be smaller than the range of float and double, but is not guaranteed to be. For decimal representations without infinities or NaN values, and with a range smaller than float, the result of a conversion from decimal to either float or double will never be infinity or NaN. end note]

#### 11.3.3 Explicit enumeration conversions

The explicit enumeration conversions are:

- From sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal to any *enum-type*.
- From any *enum-type* to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal.
- From any *enum-type* to any other *enum-type*.

An explicit enumeration conversion between two types is processed by treating any participating *enum-type* as the underlying type of that *enum-type*, and then performing an implicit or explicit numeric conversion between the resulting types. [*Example*: Given an *enum-type* E with and underlying type of int, a conversion from E to byte is processed as an explicit numeric conversion (§11.3.2) from int to byte, and a conversion from byte to E is processed as an implicit numeric conversion (§11.2.3) from byte to int. *end example*]

# 11.3.4 Explicit nullable conversions

The explicit nullable conversions are those nullable conversions (§11.6.1) derived from explicit and implicit predefined conversions.

#### 11.3.5 Explicit reference conversions

The explicit reference conversions are:

- From object and dynamic to any other *reference-type*.
- From any *class-type* S to any *class-type* T, provided S is a base class of T.
- From any *class-type* S to any *interface-type* T, provided S is not sealed and provided S does not implement T.
- From any *interface-type* S to any *class-type* T, provided T is not sealed or provided T implements S.
- From any *interface-type* S to any *interface-type* T, provided S is not derived from T.
- From an *array-type* S with an element type S<sub>E</sub> to an *array-type* T with an element type T<sub>E</sub>, provided all of the following are true:
  - S and T differ only in element type. In other words, S and T have the same number of dimensions.
  - $\circ~$  An explicit reference conversion exists from  $S_{E}$  to  $T_{E}.$
- From System. Array and the interfaces it implements, to any *array-type*.
- From a single-dimensional *array-type* S[] to System.Collections.Generic.IList<T>, System.Collections.Generic.IReadOnlyList<T>, and its base interfaces, provided that there is an identity conversion or explicit reference conversion from S to T.
- From System.Collections.Generic.IList<S>, System.Collections.Generic.IReadOnlyList<S>, and their base interfaces to a singledimensional array type T[], provided that there is an identity conversion or explicit reference conversion from S to T.
- From System.Delegate and the interfaces it implements to any *delegate-type*.
- From a reference type S to a reference type T if it has an explicit reference conversion from S to a reference type T<sub>0</sub> and T<sub>0</sub> and there is an identity conversion from T<sub>0</sub> to T.
- From a reference type S to an interface or delegate type T if it there is an explicit reference conversion from S to an interface or delegate type T<sub>0</sub> and either T<sub>0</sub> is variance-convertible to T or T is varianceconvertible to T<sub>0</sub> (§18.2.3.3).
- From  $D < S_1...S_n >$  to  $D < T_1...T_n >$  where  $D < X_1...X_n >$  is a generic delegate type,  $D < S_1...S_n >$  is not compatible with or identical to  $D < T_1...T_n >$ , and for each type parameter  $X_i$  of D the following holds:
  - $\circ$  If X<sub>i</sub> is invariant, then S<sub>i</sub> is identical to T<sub>i</sub>.
  - $\circ$  If X<sub>i</sub> is covariant, then there is an identity conversion, implicit reference conversion or explicit reference conversion from S<sub>i</sub> to T<sub>i</sub>.
  - $\circ$  If X<sub>i</sub> is contravariant, then S<sub>i</sub> and T<sub>i</sub> are either identical or both reference types.
- Explicit conversions involving type parameters that are known to be reference types. For more details on explicit conversions involving type parameters, see §11.3.8.

The explicit reference conversions are those conversions between *reference-types* that require run-time checks to ensure they are correct.

For an explicit reference conversion to succeed at run-time, the value of the source operand shall be null, or the type of the object referenced by the source operand shall be a type that can be converted to the destination type by an implicit reference conversion (§11.2.7). If an explicit reference conversion fails, a System.InvalidCastException is thrown.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. [*Note*: In other words, while a reference conversion can change the type of the reference, it never changes the type or value of the object being referred to. *end note*]

# 11.3.6 Unboxing conversions

An unboxing conversion permits a *reference-type* to be explicitly converted to a *value-type*. The following unboxing conversions exist:

- From the type object to any value-type.
- From the type System.ValueType to any value-type.
- From the type System. Enum to any *enum-type*.
- From any *interface-type* to any *non-nullable-value-type* that implements the *interface-type*.
- From any *interface-type* I to any *non-nullable-value-type* where there is an unboxing conversion from an *interface-type* I0 to the *non-nullable-value-type* and an identity conversion from I to I0.
- From any *interface-type* I to any *non-nullable-value-type* where there is an unboxing conversion from an *interface-type* I0 to the *non-nullable-value-type* and either either I0 is variance-convertible to I or I is variance-convertible to I0 (§18.2.3.3).
- From any *reference-type* to any *nullable-value-type* where there is an unboxing conversion from *reference-type* to the underlying *non-nullable-value-type* of the *nullable-value-type*.
- From a type parameter which is not known to be a value type to any type such that the conversion is permitted by §11.3.8.

An unboxing operation to a *non-nullable-value-type* consists of first checking that the object instance is a boxed value of the given *non-nullable-value-type*, and then copying the value out of the instance.

Unboxing to a *nullable-value-type* produces the null value of the *nullable-value-type* if the source operand is null, or the wrapped result of unboxing the object instance to the underlying type of the *nullable-value-type* otherwise.

[*Note*: Referring to the imaginary boxing class described in §11.2.8, an unboxing conversion of an object box to a *value-type* S consists of executing the expression ((S\_Boxing)box).value. Thus, the statements

```
object box = new S();
S s = (S)box;
```

conceptually correspond to

```
object box = new S_Boxing(new S());
S s = ((S_Boxing)box).value;
```

end note]

For an unboxing conversion to a given *non-nullable-value-type* to succeed at run-time, the value of the source operand shall be a reference to a boxed value of that *non-nullable-value-type*. If the source operand is null a System.NullReferenceException is thrown. If the source operand is a reference to an incompatible object, a System.InvalidCastException is thrown.

For an unboxing conversion to a given *nullable-value-type* to succeed at run-time, the value of the source operand shall be either null or a reference to a boxed value of the underlying *non-nullable-value-type* of the *nullable-value-type*. If the source operand is a reference to an incompatible object, a System.InvalidCastException is thrown.

# 11.3.7 Explicit dynamic conversions

An explicit dynamic conversion exists from an expression of type dynamic to any type T. The conversion is dynamically bound (§12.3.3), which means that an explicit conversion will be sought at run-time from the run-time type of the expression to T. If no conversion is found, a run-time exception is thrown.

If dynamic binding of the conversion is not desired, the expression can be first converted to object, and then to the desired type.

[Example: Assume the following class is defined:

```
class C
{
    int i;
    public C(int i) { this.i = i; }
    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

The following illustrates explicit dynamic conversions:

```
object o = "1";
dynamic d = "2";
var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

The best conversion of o to C is found at compile-time to be an explicit reference conversion. This fails at run-time, because "1" is not in fact a C. The conversion of d to C however, as an explicit dynamic conversion, is suspended to run-time, where a user defined conversion from the run-time type of d – string – to C is found, and succeeds. *end example*]

# 11.3.8 Explicit conversions involving type parameters

For a *type-parameter* T that is known to be a reference type (§15.2.5), the following explicit reference conversions (§11.3.5) exist:

- From the effective base class C of T to T and from any base class of C to T.
- From any *interface-type* to T.
- From T to any *interface-type* I provided there isn't already an implicit reference conversion from T to I.
- From a *type-parameter* U to T provided that T depends on U (§15.2.5). [*Note*: Since T is known to be a reference type, within the scope of T, the run-time type of U will always be a reference type, even if U is not known to be a reference type at compile-time. *end note*]

For a *type-parameter* T that is *not* known to be a reference type (§15.2.5), the following conversions involving T are considered to be unboxing conversions (§11.3.6) at compile-time. At run-time, if T is a value type, the conversion is executed as an unboxing conversion. At run-time, if T is a reference type, the conversion is executed as an explicit reference conversion or identity conversion.

- From the effective base class C of T to T and from any base class of C to T. [*Note*: C will be one of the types System.Object, System.ValueType, or System.Enum (otherwise T would be known to be a reference type). *end note*]
- From any *interface-type* to T.

For a *type-parameter* T that is *not* known to be a reference type (§15.2.5), the following explicit conversions exist:

- From T to any *interface-type* I provided there is not already an implicit conversion from T to I. This conversion consists of an implicit boxing conversion (§11.2.8) from T to object followed by an explicit reference conversion from object to I. At run-time, if T is a value type, the conversion is executed as a boxing conversion followed by an explicit reference conversion. At run-time, if T is a reference type, the conversion is executed as an explicit reference conversion.
- From a type parameter U to T provided that T depends on U (§15.2.5). At run-time, if T is a value type and U is a reference type, the conversion is executed as an unboxing conversion. At run-time, if both T and U are value types, then T and U are necessarily the same type and no conversion is performed. At run-time, if T is a reference type, then U is necessarily also a reference type and the conversion is executed as an explicit reference conversion or identity conversion.

In all cases, the rules ensure that a conversion is executed as an unboxing conversion if and only if at runtime the conversion is from a reference type to a value type.

The above rules do not permit a direct explicit conversion from an unconstrained type parameter to a noninterface type, which might be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear. [*Example*: Consider the following declaration:

```
class X<T>
{
    public static long F(T t) {
        return (long)t; // Error
    }
}
```

If the direct explicit conversion of t to long were permitted, one might easily expect that X<int>.F(7) would return 7L. However, it would not, because the standard numeric conversions are only considered when the types are known to be numeric at binding-time. In order to make the semantics clear, the above example must instead be written:

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t; // Ok, but will only work when T is long
    }
}
```

This code will now compile but executing X<int>.F(7) would then throw an exception at run-time, since a boxed int cannot be converted directly to a long. *end example*]

## 11.3.9 User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined explicit conversions are described in §11.5.5.

# 11.4 Standard conversions

#### 11.4.1 General

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

#### 11.4.2 Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (§11.2.2)
- Implicit numeric conversions (§11.2.3)
- Implicit nullable conversions (§11.2.5)
- Null literal conversions (§11.2.6)
- Implicit reference conversions (§11.2.7)
- Boxing conversions (§11.2.8)
- Implicit constant expression conversions (§11.2.10)
- Implicit conversions involving type parameters (§11.2.11)

The standard implicit conversions specifically exclude user-defined implicit conversions.

#### 11.4.3 Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. [*Note*: In other words, if a standard

implicit conversion exists from a type A to a type B, then a standard explicit conversion exists from type A to type B and from type B to type A. *end note*]

# 11.5 User-defined conversions

#### 11.5.1 General

C# allows the pre-defined implicit and explicit conversions to be augmented by **user-defined conversions**. User-defined conversions are introduced by declaring conversion operators (§15.10.4) in class and struct types.

#### 11.5.2 Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion.

For a given source type S and target type T, if S or T are nullable value types, let  $S_0$  and  $T_0$  refer to their underlying types, otherwise  $S_0$  and  $T_0$  are equal to S and T respectively. A class or struct is permitted to declare a conversion from a source type S to a target type T only if all of the following are true:

- S<sub>0</sub> and T<sub>0</sub> are different types.
- Either S<sub>0</sub> or T<sub>0</sub> is the class or struct type in which the operator declaration takes place.
- Neither S<sub>0</sub> nor T<sub>0</sub> is an *interface-type*.
- Excluding user-defined conversions, a conversion does not exist from S to T or from T to S.

The restrictions that apply to user-defined conversions are specified in §15.10.4.

#### 11.5.3 Evaluation of user-defined conversions

A user-defined conversion converts a *source expression*, which may have a *source type*, to another type, called the *target type*. Evaluation of a user-defined conversion centers on finding the *most-specific* user-defined conversion operator for the source expression and target type. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes, if the source type exists, along with the target type and its base classes. For this purpose it is assumed that only classes and structs can declare userdefined operators, and that non-class types have no base classes. Also, if either the source or target type is a nullable-value-type, their underlying type is used instead.
- From that set of types, determining which user-defined and lifted conversion operators are applicable. For a conversion operator to be applicable, it shall be possible to perform a standard conversion (§11.4) from the source expression to the operand type of the operator, and it shall be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most-specific. In general terms, the most-specific operator is the operator whose operand type is "closest" to the source expression and whose result type is "closest" to the target type. User-defined conversion operators are preferred over lifted conversion operators. The exact rules for establishing the most-specific user-defined conversion operator are defined in the following subclauses.

Once a most-specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source expression to the operand type of the user-defined or lifted conversion operator.
- Next, invoking the user-defined or lifted conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined or lifted conversion operator. In other words, a conversion from type S to type T will never first execute a user-defined conversion from S to X and then execute a user-defined conversion from X to T.

- Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following subclauses. The definitions make use of the following terms:
- If a standard implicit conversion (§11.4.2) exists from a type A to a type B, and if neither A nor B are *interface-types*, then A is said to be *encompassed by* B, and B is said to *encompass* A.
- If a standard implicit conversion (§11.4.2) exists from an expression E to a type B, and if neither B nor the type of E (if it has one) are *interface-types*, then E is said to be *encompassed by* B, and B is said to *encompass* E.
- The *most-encompassing type* in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most-encompassing type. In more intuitive terms, the most-encompassing type is the "largest" type in the set—the one type to which each of the other types can be implicitly converted.
- The *most-encompassed type* in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most-encompassed type. In more intuitive terms, the most-encompassed type is the "smallest" type in the set—the one type that can be implicitly converted to each of the other types.

## 11.5.4 User-defined implicit conversions

A user-defined implicit conversion from an expression E to a type T is processed as follows:

- Determine the types S, S<sub>0</sub> and T<sub>0</sub>.
  - If E has a type, let S be that type.
  - $\circ$  If S or T are nullable value types, let S<sub>U</sub> and T<sub>U</sub> be their underlying types, otherwise let S<sub>U</sub> and T<sub>U</sub> be S and T, respectively.
  - $\circ$  If S<sub>U</sub> or T<sub>U</sub> are type parameters, let S<sub>0</sub> and T<sub>0</sub> be their effective base classes, otherwise let S<sub>0</sub> and T<sub>0</sub> be S<sub>U</sub> and T<sub>U</sub>, respectively.
- Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S<sub>0</sub> (if S<sub>0</sub> exists and is a class or struct), the base classes of S<sub>0</sub> (if S<sub>0</sub> exists and is a class), and T<sub>0</sub> (if T<sub>0</sub> is a class or struct). A type is added to the set D only if an identity conversion to another type already included in the set doesn't exist.
- Find the set of applicable user-defined and lifted conversion operators, U. This set consists of the userdefined and lifted implicit conversion operators declared by the classes or structs in D that convert from a type encompassing E to a type encompassed by T. If U is empty, the conversion is undefined and a compile-time error occurs.
- Find the most-specific source type, Sx, of the operators in U:
  - $\circ$  If S exists and any of the operators in U convert from S, then S<sub>X</sub> is S.
  - Otherwise, Sx is the most-encompassed type in the combined set of source types of the operators in U. If exactly one most-encompassed type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most-specific target type, T<sub>x</sub>, of the operators in U:
  - $\circ$  If any of the operators in U convert to T, then T<sub>x</sub> is T.
  - Otherwise, Tx is the most-encompassing type in the combined set of target types of the operators in U. If exactly one most-encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most-specific conversion operator:

- If U contains exactly one user-defined conversion operator that converts from SX to TX, then this is the most-specific conversion operator.
- Otherwise, if U contains exactly one lifted conversion operator that converts from SX to TX, then this is the most-specific conversion operator.
- $\circ$   $\;$  Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
  - If E does not already have the type S<sub>x</sub>, then a standard implicit conversion from E to S<sub>x</sub> is performed.
  - $\circ$  The most-specific conversion operator is invoked to convert from S<sub>x</sub> to T<sub>x</sub>.
  - $\circ$  If T<sub>x</sub> is not T, then a standard implicit conversion from T<sub>x</sub> to T is performed.

A user-defined implicit conversion from a type S to a type T exists if a user-defined implicit conversion exists from a variable of type S to T.

#### 11.5.5 User-defined explicit conversions

A user-defined explicit conversion from an expression E to a type T is processed as follows:

- Determine the types S, S<sub>0</sub> and T<sub>0</sub>.
  - If E has a type, let S be that type.
  - $\circ~$  If S or T are nullable value types, let  $S_{u}$  and  $T_{u}$  be their underlying types, otherwise let  $S_{u}$  and  $T_{u}$  be S and T, respectively.
  - $\circ$  If S<sub>U</sub> or T<sub>U</sub> are type parameters, let S<sub>0</sub> and T<sub>0</sub> be their effective base classes, otherwise let S<sub>0</sub> and T<sub>0</sub> be S<sub>U</sub> and T<sub>U</sub>, respectively.
- Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S<sub>0</sub> (if S<sub>0</sub> exists and is a class or struct), the base classes of S<sub>0</sub> (if S<sub>0</sub> exists and is a class), T<sub>0</sub> (if T<sub>0</sub> is a class or struct), and the base classes of T<sub>0</sub> (if T<sub>0</sub> is a class). A type is added to the set D only if an identity conversion to another type already included in the set doesn't exist.
- Find the set of applicable user-defined and lifted conversion operators, U. This set consists of the userdefined and lifted implicit or explicit conversion operators declared by the classes or structs in D that convert from a type encompassing E or encompassed by S (if it exists) to a type encompassing or encompassed by T. If U is empty, the conversion is undefined and a compile-time error occurs.
- Find the most-specific source type, Sx, of the operators in U:
  - $\circ~$  If S exists and any of the operators in U convert from S, then  $S_X$  is S.
  - Otherwise, if any of the operators in U convert from types that encompass E, then Sx is the mostencompassed type in the combined set of source types of those operators. If no mostencompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, S<sub>x</sub> is the most-encompassing type in the combined set of source types of the operators in U. If exactly one most-encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most-specific target type, Tx, of the operators in U:
  - $\circ$  If any of the operators in U convert to T, then T<sub>x</sub> is T.
  - Otherwise, if any of the operators in U convert to types that are encompassed by T, then T<sub>x</sub> is the most-encompassing type in the combined set of target types of those operators. If exactly one most-encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.

- Otherwise, Tx is the most-encompassed type in the combined set of target types of the operators in U. If no most-encompassed type can be found, then the conversion is ambiguous and a compiletime error occurs.
- Find the most-specific conversion operator:
  - If U contains exactly one user-defined conversion operator that converts from S<sub>x</sub> to T<sub>x</sub>, then this is the most-specific conversion operator.
  - $\circ$  Otherwise, if U contains exactly one lifted conversion operator that converts from S<sub>x</sub> to T<sub>x</sub>, then this is the most-specific conversion operator.
  - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
  - $\circ~$  If E does not already have the type  $S_X$ , then a standard explicit conversion from E to  $S_X$  is performed.
  - $\circ$  The most-specific user-defined conversion operator is invoked to convert from S<sub>x</sub> to T<sub>x</sub>.
  - $\circ$  If T<sub>x</sub> is not T, then a standard explicit conversion from T<sub>x</sub> to T is performed.

A user-defined explicit conversion from a type S to a type T exists if a user-defined explicit conversion exists from a variable of type S to T.

## 11.6 Conversions involving nullable types

#### 11.6.1 Nullable Conversions

**Nullable conversions** permit predefined conversions that operate on non-nullable value types to also be used with nullable forms of those types. For each of the predefined implicit or explicit conversions that convert from a non-nullable value type S to a non-nullable value type T (§11.2.2, §11.2.3, §11.2.4, §11.2.10, §11.3.2 and §11.3.3), the following nullable conversions exist:

- An implicit or explicit conversion from S? to T?
- An implicit or explicit conversion from S to T?
- An explicit conversion from S? to T.

A nullable conversion is itself classified as an implicit or explicit conversion.

Certain nullable conversions are classified as standard conversions and can occur as part of a user-defined conversion. Specifically, all implicit nullable conversions are classified as standard implicit conversions (§11.4.2), and those explicit nullable conversions that satisfy the requirements of §11.4.3 are classified as standard explicit conversions.

Evaluation of a nullable conversion based on an underlying conversion from S to T proceeds as follows:

- If the nullable conversion is from S? to T?:
  - If the source value is null (HasValue property is false), the result is the null value of type T?.
  - Otherwise, the conversion is evaluated as an unwrapping from S? to S, followed by the underlying conversion from S to T, followed by a wrapping from T to T?.
- If the nullable conversion is from S to T?, the conversion is evaluated as the underlying conversion from S to T followed by a wrapping from T to T?.
- If the nullable conversion is from S? to T, the conversion is evaluated as an unwrapping from S? to S followed by the underlying conversion from S to T.

#### 11.6.2 Lifted conversions

Given a user-defined conversion operator that converts from a non-nullable value type S to a non-nullable value type T, a *lifted conversion operator* exists that converts from S? to T?. This lifted conversion operator performs an unwrapping from S? to S followed by the user-defined conversion from S to T

followed by a wrapping from T to T?, except that a null valued S? converts directly to a null valued T?. A lifted conversion operator has the same implicit or explicit classification as its underlying user-defined conversion operator.

## **11.7 Anonymous function conversions**

#### 11.7.1 General

An *anonymous-method-expression* or *lambda-expression* is classified as an anonymous function (§12.16). The expression does not have a type, but can be implicitly converted to a compatible delegate type. Some lambda expressions may also be implicitly converted to a compatible expression-tree type.

For the purpose of brevity, this subclause uses the short form for the task types Task and Task<T> (§15.15.1).

Specifically, an anonymous function F is compatible with a delegate type D provided:

- If F contains an anonymous-function-signature, then D and F have the same number of parameters.
- If F does not contain an *anonymous-function-signature*, then D may have zero or more parameters of any type, as long as no parameter of D has the out parameter modifier.
- If F has an explicitly typed parameter list, each parameter in D has the same type and modifiers as the corresponding parameter in F.
- If F has an implicitly typed parameter list, D has no ref or out parameters.
- If the body of F is an expression, and *either* D has a void return type *or* F is async and D has the return type Task, then when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid expression (w.r.t §12) that would be permitted as a *statement-expression* (§13.7).
- If the body of F is a statement block, and *either* D has a void return type *or* F is async and D has the return type Task, then when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid statement block (w.r.t §13.3) in which no return statement specifies an expression.
- If the body of F is an expression, and *either* F is non-async and D has a non-void return type T, or F is async and D has a return type Task<T>, then when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid expression (w.r.t §12) that is implicitly convertible to T.
- If the body of F is a statement block, and *either* F is non-async and D has a non-void return type T, or F is async and D has a return type Task<T>, then when each parameter of F is given the type of the corresponding parameter in D, the body of F is a valid statement block (w.r.t §13.3) with a non-reachable end point in which each return statement specifies an expression that is implicitly convertible to T.

[Example: The following examples illustrate these rules:

```
delegate void D(int x);
```

```
delegate int P(params int[] a);
                                                  // Error, end of block reachable
P p1 = delegate {
                       };
P p2 = delegate { return; };
P p3 = delegate { return 1; };
P p4 = delegate { return "Hello"; };
                                                  // Error, return type mismatch
                                                  //
                                                      ok
                                                  // Error, return type mismatch
P p5 = delegate (int[] a) {
                                                  // Ok
    return a[0];
};
  p6 = delegate(params int[] a) {
Ρ
                                                  // Error, params modifier
    return a[0];
};
P p7 = delegate(int[] a) {
    is (a length > 0) return
}
                                                  // Error, return type mismatch
    if (a.Length > 0) return a[0];
return "Hello";
};
delegate object Q(params int[] a);
Q q1 = delegate(int[] a) {
                                                  // Ok
    if (a.Length > 0) return a[0];
return "Hello";
};
```

#### end example]

[*Example*: The examples that follow use a generic delegate type Func<A, R> that represents a function that takes an argument of type A and returns a value of type R:

delegate R Func<A,R>(A arg);

In the assignments

Func <int, int=""> <math>f1 = x \Rightarrow x + 1;</math></int,>	// Ok
Func <int,double> f2 = <math>x \Rightarrow x + 1</math>;</int,double>	// Ok
Func <double, int=""> <math>f3 = x \Rightarrow x + 1;</math></double,>	// Error
Func <int, task<int="">&gt; f4 = async x =&gt; x + 1;</int,>	// Ok

the parameter and return types of each anonymous function are determined from the type of the variable to which the anonymous function is assigned.

The first assignment successfully converts the anonymous function to the delegate type Func<int,int> because, when x is given type int, x + 1 is a valid expression that is implicitly convertible to type int.

Likewise, the second assignment successfully converts the anonymous function to the delegate type Func<int,double> because the result of x + 1 (of type int) is implicitly convertible to type double.

However, the third assignment is a compile-time error because, when x is given type double, the result of x + 1 (of type double) is not implicitly convertible to type int.

The fourth assignment successfully converts the anonymous async function to the delegate type Func<int, Task<int>> because the result of x + 1 (of type int) is implicitly convertible to the effective return type int of the async lambda, which has a return type Task<int>. end example]

A lambda expression F is compatible with an expression tree type Expression<D> if F is compatible with the delegate type D. This does not apply to anonymous methods, only lambda expressions.

Certain lambda expressions cannot be converted to expression tree types: Even though the conversion *exists,* it fails at compile-time. This is the case if the lambda expression:

- Has a *block* body
- Contains simple or compound assignment operators
- Contains a dynamically bound expression
- Is async

Anonymous functions may influence overload resolution, and participate in type inference. See §12.6 for further details.

#### 11.7.2 Evaluation of anonymous function conversions to delegate types

Conversion of an anonymous function to a delegate type produces a delegate instance that references the anonymous function and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When the delegate is invoked, the body of the anonymous function is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate. A *delegate-creation-expression* (§12.7.11.6) can be used as an alternate syntax for converting an anonymous method to a delegate type.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is null, the this value of the enclosing function member, or some other object.

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable instances to the same delegate types are permitted (but not required) to return the same delegate instance. The term semantically identical is used here to mean that execution of the anonymous functions will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

```
delegate double Function(double x);
class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

Since the two anonymous function delegates have the same (empty) set of captured outer variables, and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous function expressions.

#### 11.7.3 Evaluation of anonymous function conversions to expression tree types

Conversion of an anonymous function to an expression-tree type produces an expression tree (§9.6). More precisely, evaluation of the anonymous-function conversion produces an object structure that represents the structure of the anonymous function itself. The precise structure of the expression tree, as well as the exact process for creating it, are implementation-defined.

#### 11.8 Method group conversions

An implicit conversion exists from a method group (§12.2) to a compatible delegate type (§20.4). If D is a delegate type, and E is an expression that is classified as a method group, then D is compatible with E if and only if E contains at least one method that is applicable in its normal form (§12.6.4.2) to any argument list (§12.6.2) having types and modifiers matching the parameter types and modifiers of D, as described in the following.

The compile-time application of the conversion from a method group E to a delegate type D is described in the following. Note that the existence of an implicit conversion from E to D does not guarantee that the compile-time application of the conversion will succeed without error.

- A single method M is selected corresponding to a method invocation (§12.7.6.2) of the form E(A), with the following modifications:
  - The argument list A is a list of expressions, each classified as a variable and with the type and modifier (ref or out) of the corresponding parameter in the *formal-parameter-list* of D – excepting parameters of type dynamic, where the corresponding expression has the type object instead of dynamic.
  - The candidate methods considered are only those methods that are applicable in their normal form and do not omit any optional parameters (§12.6.4.2). Thus, candidate methods are ignored if they are applicable only in their expanded form, or if one or more of their optional parameters do not have a corresponding parameter in D.
- A conversion is considered to exist if the algorithm of §12.7.6.2 produces a single best method M having the same number of parameters as D.
- Even if the conversion exists, a compile-time error occurs if the selected method M is not compatible (§20.4) with the delegate type D.
- If the selected method M is an instance method, the instance expression associated with E determines the target object of the delegate.
- If the selected method M is an extension method which is denoted by means of a member access on an instance expression, that instance expression determines the target object of the delegate.
- The result of the conversion is a value of type D, namely a delegate that refers to the selected method and target object.

[Example: The following demonstrates method group conversions:

```
delegate string D1(object o);
delegate object D2(string s);
delegate object D3();
delegate string D4(object o, params object[] a);
delegate string D5(int i);
class Test
    static string F(object o) {...}
    static void G() {
        D1 d1 = F;
                                   // Ok
        D2 d2 = F;
                                   // Ok
        \begin{array}{r} D3 & d3 = F; \\ D4 & d4 = F; \\ D5 & d5 = F; \end{array}
                                   // Error - not applicable
// Error - not applicable in normal form
// Error - applicable but not compatible
    }
}
```

The assignment to d1 implicitly converts the method group F to a value of type D1.

The assignment to d2 shows how it is possible to create a delegate to a method that has less derived (contra-variant) parameter types and a more derived (covariant) return type.

The assignment to d3 shows how no conversion exists if the method is not applicable.

The assignment to d4 shows how the method must be applicable in its normal form.

The assignment to d5 shows how parameter and return types of the delegate and method are allowed to differ only for reference types.

#### end example]

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a particular conversion. [*Example*: Thus, the example

object obj = new EventHandler(myDialog.OkClick);

could instead be written

object obj = (EventHandler)myDialog.OkClick;

#### end example]

A method group conversion can refer to a generic method, either by explicitly specifying type arguments within E, or via type inference (§12.6.3). If type inference is used, the parameter types of the delegate are used as argument types in the inference process. The return type of the delegate is not used for inference. Whether the type arguments are specified or inferred, they are part of the method group conversion process; these are the type arguments used to invoke the target method when the resulting delegate is invoked. [*Example*:

```
delegate int D(string s, int i);
delegate int E();
class X
{
   public static T F<T>(string s, T t) {...}
public static T G<T>() {...}
   static void Main() {
                           // Ok, type argument given explicitly
      D d1 = F < int >;
      D d2 = F;
                           // Ok, int inferred as type argument
                           // Ok, type argument given explicitly
      E e1 = G < int >;
                           // Error, cannot infer from return type
       E e^2 = G;
   }
}
```

#### end example]

Method groups may influence overload resolution, and participate in type inference. See §12.6 for further details.

The run-time evaluation of a method group conversion proceeds as follows:

- If the method selected at compile-time is an instance method, or it is an extension method which is accessed as an instance method, the target object of the delegate is determined from the instance expression associated with E:
  - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the instance expression is of a *reference-type*, the value computed by the instance expression becomes the target object. If the selected method is an instance method and the target object is null, a System.NullReferenceException is thrown and no further steps are executed.
  - If the instance expression is of a *value-type*, a boxing operation (§11.2.8) is performed to convert the value to an object, and this object becomes the target object.
- Otherwise, the selected method is part of a static method call, and the target object of the delegate is null.
- A delegate instance of delegate type D is obtained with a reference to the method that was determined at compile-time and a reference to the target object computed above, as follows:
  - The conversion is permitted (but not required) to use an existing delegate instance that already contains these references.

#### ECMA-334

If an existing instance was not reused, a new one is created (§20.5). If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown. Otherwise the instance is initialized with the given references.

# 12. Expressions

# 12.1 General

An expression is a sequence of operators and operands. This clause defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

# 12.2 Expression classifications

## 12.2.1 General

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left-hand side of a *member-access* (§12.7.5). In any other context, an expression classified as a namespace causes a compile-time error.
- A type. An expression with this classification can only appear as the left-hand side of a *member-access* (§12.7.5). In any other context, an expression classified as a type causes a compile-time error.
- A method group, which is a set of overloaded methods resulting from a member lookup (§12.5). A method group may have an associated instance expression and an associated type argument list. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by this (§12.7.8). A method group is permitted in an *invocation-expression* (§12.7.6) or a *delegate-creation-expression* (§12.7.11.6), and can be implicitly converted to a compatible delegate type (§11.8). In any other context, an expression classified as a method group causes a compile-time error.
- A null literal. An expression with this classification can be implicitly converted to a reference type or nullable value type
- An anonymous function. An expression with this classification can be implicitly converted to a compatible delegate type or expression tree type.
- A property access. Every property access has an associated type, namely the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the get or set block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by this (§12.7.8).
- An event access. Every event access has an associated type, namely the type of the event. Furthermore, an event access may have an associated instance expression. An event access may appear as the left-hand operand of the += and -= operators (§12.18.4). In any other context, an expression classified as an event access causes a compile-time error. When an accessor (the add or remove block) of an instance event access is invoked, the result of evaluating the instance expression becomes the instance represented by this (§12.7.8).
- An indexer access. Every indexer access has an associated type, namely the element type of the
  indexer. Furthermore, an indexer access has an associated instance expression and an associated
  argument list. When an accessor (the get or set block) of an indexer access is invoked, the result of
  evaluating the instance expression becomes the instance represented by this (§12.7.8), and the result
  of evaluating the argument list becomes the parameter list of the invocation.
- Nothing. This occurs when the expression is an invocation of a method with a return type of void. An expression classified as nothing is only valid in the context of a *statement-expression* (§13.7).

The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

A property access or indexer access is always reclassified as a value by performing an invocation of the *get-accessor* or the *set-accessor*. The particular accessor is determined by the context of the property or indexer access: If the access is the target of an assignment, the *set-accessor* is invoked to assign a new value (§12.18.2). Otherwise, the *get-accessor* is invoked to obtain the current value (§12.2.2).

An *instance accessor* is a property access on an instance, an event access on an instance, or an indexer access.

# 12.2.2 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 shall be considered definitely assigned (§10.4) 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 (§12.6.6) 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 (§12.6.6) 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.

# 12.3 Static and Dynamic Binding

# 12.3.1 General

**Binding** is the process of determining what an operation refers to, based on the type or value of expressions (arguments, operands, receivers). For instance, the binding of a method call is determined based on the type of the receiver and arguments. The binding of an operator is determined based on the type of its operands.

In C# the binding of an operation is usually determined at compile-time, based on the compile-time type of its subexpressions. Likewise, if an expression contains an error, the error is detected and reported by the compiler. This approach is known as *static binding*.

However, if an expression is a *dynamic expression* (i.e., has the type dynamic) this indicates that any binding that it participates in should be based on its run-time type rather than the type it has at compile-time. The binding of such an operation is therefore deferred until the time where the operation is to be executed during the running of the program. This is referred to as *dynamic binding*.

When an operation is dynamically bound, little or no checking is performed by the compiler. Instead if the run-time binding fails, errors are reported as exceptions at run-time.

The following operations in C# are subject to binding:

- Member access: e.M
- Method invocation: e.M(e1,...,en)
- Delegate invocation: e(e1,...,en)

- Element access: e[e1,...,en]
- Object creation: new C(e1,...,en)
- Overloaded unary operators: +, -, !, ~, ++, --, true, false
- Overloaded binary operators: +, -, \*, /, %, &, &&, |, ||, ??, ^, <<, >>, ==, !=, >, <, >=, <=
- Assignment operators: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
- Implicit and explicit conversions

When no dynamic expressions are involved, C# defaults to static binding, which means that the compiletime types of subexpressions are used in the selection process. However, when one of the subexpressions in the operations listed above is a dynamic expression, the operation is instead dynamically bound.

#### 12.3.2 Binding-time

Static binding takes place at compile-time, whereas dynamic binding takes place at run-time. In the following subclauses, the term **binding-time** refers to either compile-time or run-time, depending on when the binding takes place.

[*Example*: The following illustrates the notions of static and dynamic binding and of binding-time:

```
object o = 5;
dynamic d = 5;
Console.WriteLine(5); // static binding to Console.WriteLine(int)
Console.WriteLine(o); // static binding to Console.WriteLine(object)
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)
```

The first two calls are statically bound: the overload of Console.WriteLine is picked based on the compile-time type of their argument. Thus, the binding-time is *compile-time*.

The third call is dynamically bound: the overload of Console.WriteLine is picked based on the run-time type of its argument. This happens because the argument is a dynamic expression – its compile-time type is dynamic. Thus, the binding-time for the third call is *run-time*. *end* example]

#### 12.3.3 Dynamic binding

#### This subclause is informative.

Dynamic binding allows C# programs to interact with dynamic objects, i.e., objects that do not follow the normal rules of the C# type system. Dynamic objects may be objects from other programming languages with different types systems, or they may be objects that are programmatically setup to implement their own binding semantics for different operations.

The mechanism by which a dynamic object implements its own semantics is implementation defined. A given interface – again implementation defined – is implemented by dynamic objects to signal to the C# run-time that they have special semantics. Thus, whenever operations on a dynamic object are dynamically bound, their own binding semantics, rather than those of C# as specified in this specification, take over.

While the purpose of dynamic binding is to allow interoperation with dynamic objects, C# allows dynamic binding on all objects, whether they are dynamic or not. This allows for a smoother integration of dynamic objects, as the results of operations on them may not themselves be dynamic objects, but are still of a type unknown to the programmer at compile-time. Also, dynamic binding can help eliminate error-prone reflection-based code even when no objects involved are dynamic objects.

#### 12.3.4 Types of subexpressions

When an operation is statically bound, the type of a subexpression (e.g., a receiver, and argument, an index or an operand) is always considered to be the compile-time type of that expression.

When an operation is dynamically bound, the type of a subexpression is determined in different ways depending on the compile-time type of the subexpression:

- A subexpression of compile-time type dynamic is considered to have the type of the actual value that the expression evaluates to at run-time
- A subexpression whose compile-time type is a type parameter is considered to have the type which the type parameter is bound to at run-time
- Otherwise, the subexpression is considered to have its compile-time type.

# 12.4 Operators

# 12.4.1 General

Expressions are constructed from **operands** and **operators**. The operators of an expression indicate which operations to apply to the operands. [*Example*: Examples of operators include +, -, \*, /, and new. Examples of operands include literals, fields, local variables, and expressions. *end example*]

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 (§12.4.2).

Operands in an expression are evaluated from left to right. [*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 (§12.4.3) 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.

# 12.4.2 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*]

[Note: The following table summarizes all operators in order of precedence from highest to lowest:

Subclause	Category	Operators
§12.7	Primary	x.y f(x) a[x] x++ x new typeof default checked unchecked delegate
§12.8	Unary	+ - ! ~ ++xx (T)x await x
§12.9	Multiplicative	* / %
§12.9	Additive	+ -
§12.10	Shift	<< >>
§12.11	Relational and type-testing	< > <= >= is as
§12.11	Equality	== !=
§12.12	Logical AND	&
§12.12	Logical XOR	^
§12.12	Logical OR	
§12.13	Conditional AND	&&
§12.13	Conditional OR	11
§12.14	Null coalescing	??
§12.15	Conditional	?:
§12.18 and §12.16	Assignment and lambda expression	= *= /= %= += -= <<= >>= &= ^=  = =>

# end note]

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 and the null coalescing operator, all binary operators are *left-associative*, meaning that operations are performed from left to right. [*Example*: x + y + z is evaluated as (x + y) + z. *end example*]
- The assignment operators, the null coalescing operator and the conditional operator (?:) are *right-associative*, meaning that operations are performed from right to left. [*Example*: x = y = z is evaluated as x = (y = z). *end example*]

Precedence and associativity can be controlled using parentheses. [*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*]

# 12.4.3 Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including operator declarations (§15.10) in classes and structs. User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operators implementations exist will the predefined operator implementations be considered, as described in §12.4.4 and §12.4.5.

#### The overloadable unary operators are:

+ - ! ~ ++ -- true false

[Note: Although true and false are not used explicitly in expressions (and therefore are not included in the precedence table in §12.4.2), they are considered operators because they are invoked in several expression contexts: Boolean expressions (§12.21) and expressions involving the conditional (§12.15) and conditional logical operators (§12.13). *end note*]

The *overloadable binary operators* are:

+ - \* / % & | ^ << >> == != > < >= <=

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the =, &&, ||, ??, ?:, =>, checked, unchecked, new, typeof, default, as, and is operators.

When a binary operator is overloaded, the corresponding compound assignment operator, if any, is also implicitly overloaded. [*Example*: An overload of operator \* is also an overload of operator \*=. This is described further in §12.18. *end example*] The assignment operator itself (=) cannot be overloaded. An assignment always performs a simple store of a value into a variable(§12.18.2).

Cast operations, such as (T)x, are overloaded by providing user-defined conversions (§11.5). [*Note:* User-defined conversions do not affect the behavior of the *i*s or as operators. *end note*]

Element access, such as a[x], is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (§15.9).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix ++ and -- operators. In the third entry, *op* denotes any overloadable binary operator. [*Note*: For an example of overloading the ++ and -- operators see §15.10.2. *end note*]

Operator notation	Functional notation
ор х	operator op(x)
х ор	operator op(x)
х ор у	operator op(x, y)

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. [*Note*: Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator. *end note*]

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. [*Example*: The / operator is always a binary operator, always has the precedence level specified in §12.4.2, and is always left-associative. *end example*]

[*Note*: While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of operator == should compare the two operands for equality and return an appropriate bool result. *end note*]

The descriptions of individual operators in §12.8 through §12.18 specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms *unary operator overload resolution, binary operator overload resolution, numeric promotion,* and *lifted operators* definitions of which are found in the following subclauses.

## 12.4.4 Unary operator overload resolution

An operation of the form op x or x op, where op is an overloadable unary operator, and x is an expression of type X, is processed as follows:

- The set of candidate user-defined operators provided by X for the operation operator *op*(x) is determined using the rules of §12.4.6.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary operator *op* implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator. The predefined operators provided by an enum or delegate type are only included in this set when the binding-time type—or the underlying type if it is a nullable type—of either operand is the enum or delegate type.
- The overload resolution rules of §12.6.4 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 binding-time error occurs.

#### 12.4.5 Binary operator overload resolution

An operation of the form x *op* y, where *op* is an overloadable binary operator, x is an expression of type X, and y is an expression of type Y, is processed as follows:

- The set of candidate user-defined operators provided by X and Y for the operation operator op(x, y) is determined. The set consists of the union of the candidate operators provided by X and the candidate operators provided by Y, each determined using the rules of §12.4.6. For the combined set, candidates are merged as follows:
  - 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 there is an identity conversion between X and Y, an operator *op*Y provided by Y has the same return type as an *op*X provided by X and the operand types of *op*Y have an identity conversion to the corresponding operand types of *op*X then only *op*X occurs in the set.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary operator op implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator. For predefined enum and delegate operators, the only operators considered are those provided by an enum or delegate type that is the binding-time type of one of the operands.
- The overload resolution rules of §12.6.4 are applied to the set of candidate operators to select the best operator with respect to the argument list (x, y), and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

#### 12.4.6 Candidate user-defined operators

Given a type T and an operation operator op(A), where op is an overloadable operator and A is an argument list, the set of candidate user-defined operators provided by T for operator op(A) is determined as follows:

- Determine the type  $T_0$ . If T is a nullable value type,  $T_0$  is its underlying type; otherwise,  $T_0$  is equal to T.
- For all operator *op* declarations in T<sub>0</sub> and all lifted forms of such operators, if at least one operator is applicable (§12.6.4.2) with respect to the argument list A, then the set of candidate operators consists of all such applicable operators in T<sub>0</sub>.
- Otherwise, if T<sub>0</sub> is object, the set of candidate operators is empty.

• Otherwise, the set of candidate operators provided by T<sub>0</sub> is the set of candidate operators provided by the direct base class of T<sub>0</sub>, or the effective base class of T<sub>0</sub> if T<sub>0</sub> is a type parameter.

# 12.4.7 Numeric promotions

#### 12.4.7.1 General

#### This subclause 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 (\$12.6.4) 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 the operation b \* s, where b is a byte and s is a short, overload resolution selects operator \*(int, int) as the best operator. Thus, the effect is that b and s are converted to int, and the type of the result is int. Likewise, for the operation i \* d, where i is an int and d is a double, overload resolution selects operator \*(double, double) as the best operator. *end example*]

#### End of informative text.

#### 12.4.7.2 Unary numeric promotions

#### This subclause is informative.

Unary numeric promotion occurs for the operands of the predefined +, -, and ~ unary operators. Unary numeric promotion simply consists of converting operands of type sbyte, byte, short, ushort, or char to type int. Additionally, for the unary - operator, unary numeric promotion converts operands of type uint to type long.

#### End of informative text.

#### 12.4.7.3 Binary numeric promotions

#### This subclause is informative.

Binary numeric promotion occurs for the operands of the predefined +, -, \*, /, %, &,  $|, \wedge, ==, !=, >, <, >=,$  and <= binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rules, in the order they appear here:

- If either operand is of type decimal, the other operand is converted to type decimal, or a bindingtime error occurs if the other operand is of type float or double.
- Otherwise, if either operand is of type double, the other operand is converted to type double.
- Otherwise, if either operand is of type float, the other operand is converted to type float.
- Otherwise, if either operand is of type ulong, the other operand is converted to type ulong, or a binding-time error occurs if the other operand is of type sbyte, short, int, or long.
- Otherwise, if either operand is of type long, the other operand is converted to type long.

- Otherwise, if either operand is of type uint and the other operand is of type sbyte, short, or int, both operands are converted to type long.
- Otherwise, if either operand is of type uint, the other operand is converted to type uint.
- Otherwise, both operands are converted to type int.

[*Note*: The first rule disallows any operations that mix the decimal type with the double and float types. The rule follows from the fact that there are no implicit conversions between the decimal type and the double and float types. *end note*]

[*Note*: Also note that it is not possible for an operand to be of type ulong when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of ulong as well as the signed integral types. *end note*]

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

[Example: In the following code

```
decimal AddPercent(decimal x, double percent) {
   return x * (1.0 + percent / 100.0);
}
```

a binding-time error occurs because a decimal cannot be multiplied by a double. The error is resolved by explicitly converting the second operand to decimal, as follows:

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

end example]

End of informative text.

#### 12.4.8 Lifted operators

*Lifted operators* permit predefined and user-defined operators that operate on non-nullable value types to also be used with nullable forms of those types. Lifted operators are constructed from predefined and user-defined operators that meet certain requirements, as described in the following:

• For the unary operators

+ ++ - -- ! ~

a lifted form of an operator exists if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single ? modifier to the operand and result types. The lifted operator produces a null value if the operand is null. Otherwise, the lifted operator unwraps the operand, applies the underlying operator, and wraps the result.

• For the binary operators

+ - \* / % & | ^ << >>

a lifted form of an operator exists if the operand and result types are all non-nullable value types. The lifted form is constructed by adding a single ? modifier to each operand and result type. The lifted operator produces a null value if one or both operands are null (an exception being the & and | operators of the bool? type, as described in §12.12.5). Otherwise, the lifted operator unwraps the operands, applies the underlying operator, and wraps the result.

• For the equality operators

== !=

a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is bool. The lifted form is constructed by adding a single ? modifier to each operand type. The lifted operator considers two null values equal, and a null value unequal to any non-null value. If both operands

are non-null, the lifted operator unwraps the operands and applies the underlying operator to produce the bool result.

- For the relational operators
  - < > <= >=

a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is bool. The lifted form is constructed by adding a single ? modifier to each operand type. The lifted operator produces the value false if one or both operands are null. Otherwise, the lifted operator unwraps the operands and applies the underlying operator to produce the bool result.

## 12.5 Member lookup

#### 12.5.1 General

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup can occur as part of evaluating a *simple-name* (§12.7.3) or a *member-access* (§12.7.5) in an expression. If the *simple-name* or *member-access* occurs as the *primary-expression* of an *invocation-expression* (§12.7.6.2), the member is said to be *invoked*.

If a member is a method or event, or if it is a constant, field or property of either a delegate type (§20) or the type dynamic (§9.2.4), then the member is said to be *invocable*.

Member lookup considers not only the name of a member but also the number of type parameters the member has and whether the member is accessible. For the purposes of member lookup, generic methods and nested generic types have the number of type parameters indicated in their respective declarations and all other members have zero type parameters.

A member lookup of a name N with K type arguments in a type T is processed as follows:

- First, a set of accessible members named N is determined:
  - If T is a type parameter, then the set is the union of the sets of accessible members named N in each of the types specified as a primary constraint or secondary constraint (§15.2.5) for T, along with the set of accessible members named N in object.
  - Otherwise, the set consists of all accessible (§8.5) members named N in T, including inherited members and the accessible members named N in object. If T is a constructed type, the set of members is obtained by substituting type arguments as described in §15.3.3. Members that include an override modifier are excluded from the set.
- Next, if K is zero, all nested types whose declarations include type parameters are removed. If K is not zero, all members with a different number of type parameters are removed. When K is zero, methods having type parameters are not removed, since the type inference process (§12.6.3) might be able to infer the type arguments.
- Next, if the member is invoked, all non-invocable members are removed from the set.
- Next, members that are hidden by other members are removed from the set. For every member S.M in the set, where S is the type in which the member M is declared, the following rules are applied:
  - If M is a constant, field, property, event, or enumeration member, then all members declared in a base type of S are removed from the set.
  - If M is a type declaration, then all non-types declared in a base type of S are removed from the set, and all type declarations with the same number of type parameters as M declared in a base type of S are removed from the set.
  - If M is a method, then all non-method members declared in a base type of S are removed from the set.
- Next, interface members that are hidden by class members are removed from the set. This step only has an effect if T is a type parameter and T has both an effective base class other than object and a

non-empty effective interface set (§15.2.5). For every member S.M in the set, where S is the type in which the member M is declared, the following rules are applied if S is a class declaration other than object:

- If M is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
- If M is a method, then all non-method members declared in an interface declaration are removed from the set, and all methods with the same signature as M declared in an interface declaration are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined:
  - If the set consists of a single member that is not a method, then this member is the result of the lookup.
  - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
  - Otherwise, the lookup is ambiguous, and a binding-time error occurs.

For member lookups in types other than type parameters and interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in §18.4.6. [*Note*: This phase only accounts for one kind of ambiguity. If the member lookup results in a method group, further uses of method group may fail due to ambiguity, for example as described in §12.6.4.1 and §12.6.6.2. *end note*]

## 12.5.2 Base types

For purposes of member lookup, a type T is considered to have the following base types:

- If T is object or dynamic, then T has no base type.
- If T is an *enum-type*, the base types of T are the class types System.Enum, System.ValueType, and object.
- If T is a struct-type, the base types of T are the class types System.ValueType and object. [Note: A nullable-value-type is a struct-type (§9.3.1). end note]
- If T is a *class-type*, the base types of T are the base classes of T, including the class type object.
- If T is an *interface-type*, the base types of T are the base interfaces of T and the class type object.
- If T is an *array-type*, the base types of T are the class types System.Array and object.
- If T is a *delegate-type*, the base types of T are the class types System.Delegate and object.

#### **12.6 Function members**

#### 12.6.1 General

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Finalizers

Except for finalizers and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list (§12.6.2) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of generic methods may employ type inference to determine the set of type arguments to pass to the method. This process is described in §12.6.3.

Invocations of methods, indexers, operators, and instance constructors employ overload resolution to determine which of a candidate set of function members to invoke. This process is described in §12.6.4.

Once a particular function member has been identified at binding-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in §12.6.6.

[*Note*: The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the table, e, x, y, and value indicate expressions classified as variables or values, T indicates an expression classified as a type, F is the simple name of a method, and P is the simple name of a property.

Construct	Example	Description
Method invocation	F(x,y)	Overload resolution is applied to select the best method F in the containing class or struct. The method is invoked with the argument list $(x, y)$ . If the method is not static, the instance expression is this.
	T.F(x,y)	Overload resolution is applied to select the best method F in the class or struct T. A binding-time error occurs if the method is not static. The method is invoked with the argument list $(x, y)$ .
	e.F(x,y)	Overload resolution is applied to select the best method F in the class, struct, or interface given by the type of e. A binding- time error occurs if the method is static. The method is invoked with the instance expression e and the argument list (x, y).
Property access	P	The get accessor of the property P in the containing class or struct is invoked. A compile-time error occurs if P is write-only. If P is not static, the instance expression is this.
	P = value	The set accessor of the property P in the containing class or struct is invoked with the argument list (value). A compile- time error occurs if P is read-only. If P is not static, the instance expression is this.
	Т.Р	The get accessor of the property P in the class or struct T is invoked. A compile-time error occurs if P is not static or if P is write-only.
	T.P=value	The set accessor of the property P in the class or struct T is invoked with the argument list (value). A compile-time error occurs if P is not static or if P is read-only.

Construct	Example	Description		
	e.P	The get accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e. A binding-time error occurs if P is static or if P is write-only.		
	e.P=value	The set accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e and the argument list (value). A binding-time error occurs if P is static or if P is read-only.		
Event access	E += value	The add accessor of the event E in the containing class or struct is invoked. If E is not static, the instance expression is this.		
	E -= value	The remove accessor of the event E in the containing class or struct is invoked. If E is not static, the instance expression is this.		
	T.E += value	The add accessor of the event E in the class or struct T is invoked. A binding-time error occurs if E is not static.		
	T.E-=value	The remove accessor of the event E in the class or struct T is invoked. A binding-time error occurs if E is not static.		
	e.E+=value	The add accessor of the event E in the class, struct, or interface given by the type of e is invoked with the instance expression e. A binding-time error occurs if E is static.		
	e.E-=value	The remove accessor of the event E in the class, struct, or interface given by the type of e is invoked with the instance expression e. A binding-time error occurs if E is static.		
Indexer access	e[x,y]	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of e. The get accessor of the indexer is invoked with the instance expression e and the argument list (x, y). A binding-time error occurs if the indexer is write-only.		
	e[x,y]=value	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of e. The set accessor of the indexer is invoked with the instance expression e and the argument list (x, y, value). A binding-time error occurs if the indexer is read-only.		
Operator invocation	-x	Overload resolution is applied to select the best unary operator in the class or struct given by the type of x. The selected operator is invoked with the argument list (x).		
	x + y	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of x and y. The selected operator is invoked with the argument list $(x, y)$ .		
Instance constructor invocation	new T(x, y)	Overload resolution is applied to select the best instance constructor in the class or struct T. The instance constructor is invoked with the argument list (x, y).		

end note]

# 12.6.2 Argument lists

### 12.6.2.1 General

Every function member and delegate invocation includes an argument list, which provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For instance constructors, methods, indexers and delegates, the arguments are specified as an *argument-list*, as described below. For indexers, when invoking the set accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator. [*Note*: This additional argument is not used for overload resolution, just during invocation of the set accessor. *end note*]
- For properties, the argument list is empty when invoking the get accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the set accessor.
- For events, the argument list consists of the expression specified as the right operand of the += or = operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

The arguments of properties (§15.7), events (§15.8), and user-defined operators (§15.10) are always passed as value parameters (§15.6.2.2). The arguments of indexers (§15.9) are always passed as value parameters (§15.6.2.2) or parameter arrays (§15.6.2.5). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, indexer, or delegate invocation are specified as an *argument-list*:

argument-list: argument argument-list , argument argument: argument-name<sub>opt</sub> argument-value argument-name: identifier : argument-value: expression ref variable-reference out variable-reference

An *argument-list* consists of one or more *arguments*, separated by commas. Each argument consists of an optional *argument-name* followed by an *argument-value*. An *argument* with an *argument-name* is referred to as a *named argument*, whereas an *argument* without an *argument-name* is a *positional argument*. It is an error for a positional argument to appear after a named argument in an *argument-list*.

The *argument-value* can take one of the following forms:

- An expression, indicating that the argument is passed as a value parameter (§15.6.2.2).
- The keyword ref followed by a *variable-reference* (§10.5), indicating that the argument is passed as a reference parameter (§15.6.2.3). A variable shall be definitely assigned (§10.4) before it can be passed as a reference parameter.

• The keyword out followed by a *variable-reference* (§10.5), indicating that the argument is passed as an output parameter (§15.6.2.4). A variable is considered definitely assigned (§10.4) following a function member invocation in which the variable is passed as an output parameter.

The form determines the *parameter-passing mode* of the argument: *value, reference,* or *output,* respectively.

Passing a volatile field (§15.5.4) as a reference parameter or output parameter causes a warning, since the field may not be treated as volatile by the invoked method.

#### 12.6.2.2 Corresponding parameters

For each argument in an argument list there has to be a corresponding parameter in the function member or delegate being invoked.

The parameter list used in the following is determined as follows:

- For virtual methods and indexers defined in classes, the parameter list is picked from the first declaration or override of the function member found when starting with the static type of the receiver, and searching through its base classes.
- For partial methods, the parameter list of the defining partial method declaration is used.
- For all other function members and delegates there is only a single parameter list, which is the one used.

The position of an argument or parameter is defined as the number of arguments or parameters preceding it in the argument list or parameter list.

The corresponding parameters for function member arguments are established as follows:

- Arguments in the *argument-list* of instance constructors, methods, indexers and delegates:
  - A positional argument where a parameter occurs at the same position in the parameter list corresponds to that parameter, unless the parameter is a parameter array and the function member is invoked in its expanded form.
  - A positional argument of a function member with a parameter array invoked in its expanded form, which occurs at or after the position of the parameter array in the parameter list, corresponds to an element in the parameter array.
  - A named argument corresponds to the parameter of the same name in the parameter list.
  - For indexers, when invoking the set accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit value parameter of the set accessor declaration.
- For properties, when invoking the get accessor there are no arguments. When invoking the set accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit value parameter of the set accessor declaration.
- For user-defined unary operators (including conversions), the single operand corresponds to the single parameter of the operator declaration.
- For user-defined binary operators, the left operand corresponds to the first parameter, and the right operand corresponds to the second parameter of the operator declaration.

# 12.6.2.3 Run-time evaluation of argument lists

During the run-time processing of a function member invocation (§12.6.6), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

• For a value parameter, the argument expression is evaluated and an implicit conversion (§11.2) to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.

• For a reference or output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation. If the variable reference given as a reference or output parameter is an array element of a *reference-type*, a run-time check is performed to ensure that the element type of the array is identical to the type of the parameter. If this check fails, a System.ArrayTypeMismatchException is thrown.

Methods, indexers, and instance constructors may declare their right-most parameter to be a parameter array (§15.6.2.5). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (§12.6.4.2):

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array shall be a single expression that is implicitly convertible (§11.2) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation shall specify zero or more positional arguments for the parameter array, where each argument is an expression that is implicitly convertible (§11.2) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in textual order. [Example: Thus, the example

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }
    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

produces the output

x = 0, y = 1, z = 2x = 4, y = -1, z = 3

end example]

The array co-variance rules (§17.6) permit a value of an array type A[] to be a reference to an instance of an array type B[], provided an implicit reference conversion exists from B to A. Because of these rules, when an array element of a *reference-type* is passed as a reference or output parameter, a run-time check is required to ensure that the actual element type of the array is *identical* to that of the parameter. [*Example*: In the following code

```
class Test
{
   static void F(ref object x) {...}
   static void Main() {
      object[] a = new object[10];
      object[] b = new string[10];
      F(ref a[0]); // Ok
      F(ref b[1]); // ArrayTypeMismatchException
   }
}
```

the second invocation of F causes a System.ArrayTypeMismatchException to be thrown because the actual element type of b is string and not object. *end example*]

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (§12.7.11.5) was inserted around the expanded parameters. [*Example*: Given the declaration

void F(int x, int y, params object[] args);

the following invocations of the expanded form of the method

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspond exactly to

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

In particular, note that an empty array is created when there are zero arguments given for the parameter array. *end example*]

When arguments are omitted from a function member with corresponding optional parameters, the default arguments of the function member declaration are implicitly passed. [*Note*: Because these are always constant, their evaluation will not impact the evaluation of the remaining arguments. *end note*]

#### 12.6.3 Type inference

#### 12.6.3.1 General

When a generic method is called without specifying type arguments, a **type inference** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. [*Example*: Given the method declaration:

```
class Chooser
{
   static Random rand = new Random();
   public static T Choose<T>(T first, T second) {
      return (rand.Next(2) == 0)? first: second;
   }
}
```

it is possible to invoke the Choose method without explicitly specifying a type argument:

<pre>int i = Chooser.Choose(5, 213);</pre>	// Calls Choose <int></int>
<pre>string s = Chooser.Choose("foo", "bar");</pre>	// Calls Choose <string></string>

Through type inference, the type arguments int and string are determined from the arguments to the method. *end example*]

Type inference occurs as part of the binding-time processing of a method invocation (§12.7.6.2) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a binding-time error. However, it often leads to a binding-time error when overload resolution then fails to find any applicable methods.

If each supplied argument does not correspond to exactly one parameter in the method (§12.6.2.2), or there is a non-optional parameter with no corresponding argument, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

 $T_r M < X_1 \dots X_n > (T_1 p_1 \dots T_m p_m)$ 

With a method call of the form  $M(E_1 ... E_m)$  the task of type inference is to find unique type arguments  $S_{1...}S_n$  for each of the type parameters  $X_{1...}X_n$  so that the call  $M < S_{1...}S_n > (E_{1...}E_m)$  becomes valid.

The process of type inference is described below as an algorithm. A conformant compiler may be implemented using an alternative approach, provided it reaches the same result in all cases.

During the process of inference each type parameter  $X_i$  is either *fixed* to a particular type  $S_i$  or *unfixed* with an associated set of *bounds*. Each of the bounds is some type T. Initially each type variable  $X_i$  is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

[*Note:* Type inference takes place not only when a generic method is called. Type inference for conversion of method groups is described in §12.6.3.14 and finding the best common type of a set of expressions is described in §12.6.3.15. *end note*]

# 12.6.3.2 The first phase

For each of the method arguments E<sub>1</sub>:

- If Ei is an anonymous function, an *explicit parameter type inference* (§12.6.3.8) is made *from* Ei to Ti
- Otherwise, if E<sub>1</sub> has a type U and x<sub>1</sub> is a value parameter (§15.6.2.2) then a *lower-bound inference* (§12.6.3.10) is made *from* U to T<sub>1</sub>.
- Otherwise, if E<sub>i</sub> has a type U and x<sub>i</sub> is a reference (§15.6.2.3) or output (§15.6.2.4) parameter then an *exact inference* (§12.6.3.9) is made *from* U to T<sub>i</sub>.
- Otherwise, no inference is made for this argument.

# 12.6.3.3 The second phase

The second phase proceeds as follows:

- All unfixed type variables X<sub>i</sub> which do not depend on (§12.6.3.6) any X<sub>j</sub> are fixed (§12.6.3.12).
- If no such type variables exist, all *unfixed* type variables X<sub>1</sub> are *fixed* for which all of the following hold:
  - $\circ$   $\;$  There is at least one type variable  $x_{j}\;$  that depends on  $x_{i}\;$
  - $\circ \quad X_i \text{ has a non-empty set of bounds} \\$
- If no such type variables exist and there are still *unfixed* type variables, type inference fails.
- Otherwise, if no further *unfixed* type variables exist, type inference succeeds.
- Otherwise, for all arguments E<sub>i</sub> with corresponding parameter type T<sub>i</sub> where the *output types* (§12.6.3.5) contain *unfixed* type variables X<sub>j</sub> but the *input types* (§12.6.3.4) do not, an *output type inference* (§12.6.3.7) is made *from* E<sub>i</sub> to T<sub>i</sub>. Then the second phase is repeated.

# 12.6.3.4 Input types

If E is a method group or implicitly typed anonymous function and T is a delegate type or expression tree type then all the parameter types of T are *input types of* E *with type* T.

# 12.6.3.5 Output types

If E is a method group or an anonymous function and T is a delegate type or expression tree type then the return type of T is an *output type of* E *with type* T.

# 12.6.3.6 Dependence

An *unfixed* type variable  $X_i$  *depends directly on* an *unfixed* type variable  $X_j$  if for some argument  $E_k$  with type  $T_k X_j$  occurs in an *input type* of  $E_k$  with type  $T_k$  and  $X_i$  occurs in an *output type* of  $E_k$  with type  $T_k$ .

 $X_j$  depends on  $X_i$  if  $X_j$  depends directly on  $X_i$  or if  $X_i$  depends directly on  $X_k$  and  $X_k$  depends on  $X_j$ . Thus "depends on" is the transitive but not reflexive closure of "depends directly on".

### 12.6.3.7 Output type inferences

An *output type inference* is made *from* an expression E *to* a type T in the following way:

- If E is an anonymous function with inferred return type U (§12.6.3.13) and T is a delegate type or expression tree type with return type T<sub>b</sub>, then a *lower-bound inference* (§12.6.3.10) is made *from* U to T<sub>b</sub>.
- Otherwise, if E is a method group and T is a delegate type or expression tree type with parameter types T<sub>1</sub>...T<sub>k</sub> and return type T<sub>b</sub>, and overload resolution of E with the types T<sub>1</sub>...T<sub>k</sub> yields a single method with return type U, then a *lower-bound inference* is made *from* U to T<sub>b</sub>.
- Otherwise, if E is an expression with type U, then a *lower-bound inference* is made from U to T.
- Otherwise, no inferences are made.

#### 12.6.3.8 Explicit parameter type inferences

An *explicit parameter type inference* is made *from* an expression E *to* a type T in the following way:

• If E is an explicitly typed anonymous function with parameter types U<sub>1</sub>...U<sub>k</sub> and T is a delegate type or expression tree type with parameter types V<sub>1</sub>...V<sub>k</sub> then for each U<sub>1</sub> an *exact inference* (§12.6.3.9) is made *from* U<sub>1</sub> *to* the corresponding V<sub>1</sub>.

#### 12.6.3.9 Exact inferences

An *exact inference from* a type U to a type V is made as follows:

- If V is one of the *unfixed* X<sub>1</sub> then U is added to the set of exact bounds for X<sub>1</sub>.
- Otherwise, sets  $V_{1...}V_k$  and  $U_{1...}U_k$  are determined by checking if any of the following cases apply:
  - $\circ$  V is an array type V<sub>1</sub>[...] and U is an array type U<sub>1</sub>[...] of the same rank
  - $\circ$  V is the type V<sub>1</sub>? and U is the type U<sub>1</sub>?
  - $\circ~~V$  is a constructed type C<V1...Vk> and ~~U is a constructed type C<U1...Uk>
  - If any of these cases apply then an *exact inference* is made from each U<sub>i</sub> to the corresponding V<sub>i</sub>.
- Otherwise, no inferences are made.

#### 12.6.3.10 Lower-bound inferences

A lower-bound inference from a type U to a type V is made as follows:

- If V is one of the *unfixed* X<sub>1</sub> then U is added to the set of lower bounds for X<sub>1</sub>.
- Otherwise, if V is the type V<sub>1</sub>? and U is the type U<sub>1</sub>? then a lower bound inference is made from U<sub>1</sub> to V<sub>1</sub>.
- Otherwise, sets  $U_1...U_k$  and  $V_1...V_k$  are determined by checking if any of the following cases apply:
  - $\circ$  V is an array type V<sub>1</sub>[...] and U is an array type U<sub>1</sub>[...] of the same rank

- V is one of IEnumerable<V<sub>1</sub>>, ICollection<V<sub>1</sub>>, IReadOnlyList<V<sub>1</sub>>>, IReadOnlyCollection<V<sub>1</sub>> or IList<V<sub>1</sub>> and U is a single-dimensional array type U<sub>1</sub>[]
- V is a constructed class, struct, interface or delegate type  $C < V_1...V_k >$  and there is a unique type  $C < U_1...U_k >$  such that U (or, if U is a type parameter, its effective base class or any member of its effective interface set) is identical to, inherits from (directly or indirectly), or implements (directly or indirectly)  $C < U_1...U_k >$ .
- (The "uniqueness" restriction means that in the case interface C<T>{} class U: C<X>,
   C<Y>{}, then no inference is made when inferring from U to C<T> because U1 could be X or Y.)

If any of these cases apply then an inference is made from each  $U_i$  to the corresponding  $V_i$  as follows:

- If U<sub>1</sub> is not known to be a reference type then an *exact inference* is made
- Otherwise, if U is an array type then a *lower-bound inference* is made
- $\circ$  Otherwise, if V is C<V<sub>1...</sub>V<sub>k</sub>> then inference depends on the i-th type parameter of C:
  - If it is covariant then a *lower-bound inference* is made.
  - If it is contravariant then an *upper-bound inference* is made.
  - If it is invariant then an *exact inference* is made.
- Otherwise, no inferences are made.

#### 12.6.3.11 Upper-bound inferences

An *upper-bound inference from* a type U to a type V is made as follows:

- If V is one of the *unfixed* X<sub>i</sub> then U is added to the set of upper bounds for X<sub>i</sub>.
- Otherwise, sets V<sub>1</sub>...V<sub>k</sub> and U<sub>1</sub>...U<sub>k</sub> are determined by checking if any of the following cases apply:
  - $\circ$  U is an array type U<sub>1</sub>[...] and V is an array type V<sub>1</sub>[...] of the same rank
  - U is one of IEnumerable<U<sub>e</sub>>, ICollection<U<sub>e</sub>>, IReadOnlyList<U<sub>e</sub>>, IReadOnlyCollection<U<sub>e</sub>> or IList<U<sub>e</sub>> and V is a single-dimensional array type V<sub>e</sub>[]
  - $\circ$  U is the type U<sub>1</sub>? and V is the type V<sub>1</sub>?
  - U is constructed class, struct, interface or delegate type C<U1...Uk> and V is a class, struct, interface or delegate type which is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) a unique type C<V1...Vk>
  - (The "uniqueness" restriction means that if we have interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}, then no inference is made when inferring from C<U1> to V<Q>.
     Inferences are not made from U1 to either X<Q> or Y<Q>.)

If any of these cases apply then an inference is made from each U<sub>1</sub> to the corresponding V<sub>1</sub> as follows:

- If U<sub>1</sub> is not known to be a reference type then an *exact inference* is made
- Otherwise, if V is an array type then an *upper-bound inference* is made
- $\circ$  Otherwise, if U is C<U<sub>1</sub>...U<sub>k</sub>> then inference depends on the i-th type parameter of C:
  - If it is covariant then an *upper-bound inference* is made.
  - If it is contravariant then a *lower-bound inference* is made.
  - If it is invariant then an *exact inference* is made.
- Otherwise, no inferences are made.

#### 12.6.3.12 Fixing

An *unfixed* type variable  $X_i$  with a set of bounds is *fixed* as follows:

• The set of candidate types U<sub>j</sub> starts out as the set of all types in the set of bounds for X<sub>i</sub>.

- We then examine each bound for X<sub>i</sub> in turn: For each exact bound U of X<sub>i</sub> all types U<sub>j</sub> that are not identical to U are removed from the candidate set. For each lower bound U of X<sub>i</sub> all types U<sub>j</sub> to which there is *not* an implicit conversion from U are removed from the candidate set. For each upper-bound U of X<sub>i</sub> all types U<sub>j</sub> from which there is *not* an implicit conversion to U are removed from the candidate set.
- If among the remaining candidate types U<sub>j</sub> there is a unique type V to which there is an implicit conversion from all the other candidate types, then X<sub>i</sub> is fixed to V.
- Otherwise, type inference fails.

# 12.6.3.13 Inferred return type

The inferred return type of an anonymous function F is used during type inference and overload resolution. The inferred return type can only be determined for an anonymous function where all parameter types are known, either because they are explicitly given, provided through an anonymous function conversion or inferred during type inference on an enclosing generic method invocation.

The *inferred effective return type* is determined as follows:

- If the body of F is an *expression* that has a type, then the inferred effective return type of F is the type of that expression.
- If the body of F is a *block* and the set of expressions in the block's return statements has a best common type T (§12.6.3.15), then the inferred effective return type of F is T.
- Otherwise, an effective return type cannot be inferred for F.

The *inferred return type* is determined as follows:

- If F is async and the body of F is either an expression classified as nothing (§12.2), or a statement block where no return statements have expressions, the inferred return type is System.Threading.Tasks.Task
- If F is async and has an inferred effective return type T, the inferred return type is System.Threading.Tasks.Task<T>.
- If F is non-async and has an inferred effective return type T, the inferred return type is T.
- Otherwise, a return type cannot be inferred for F.

[*Example*: As an example of type inference involving anonymous functions, consider the Select extension method declared in the System.Linq.Enumerable class:

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource,TResult>(
            this IEnumerable<TSource> source,
            Func<TSource,TResult> selector)
        {
            foreach (TSource element in source) yield return
        selector(element);
        }
    }
}
```

Assuming the System.Linq namespace was imported with a using namespace directive, and given a class Customer with a Name property of type string, the Select method can be used to select the names of a list of customers:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

The extension method invocation (§12.7.6.3) of Select is processed by rewriting the invocation to a static method invocation:

IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the customers argument is related to the source parameter, inferring TSource to be Customer. Then, using the anonymous function type inference process described above, c is given type Customer, and the expression c.Name is related to the return type of the selector parameter, inferring TResult to be string. Thus, the invocation is equivalent to

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

and the result is of type IEnumerable<string>.

The following example demonstrates how anonymous function type inference allows type information to "flow" between arguments in a generic method invocation. Given the method:

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
    return f2(f1(value));
}
```

Type inference for the invocation:

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t =>
t.TotalSeconds);
```

proceeds as follows: First, the argument "1:15:30" is related to the value parameter, inferring X to be string. Then, the parameter of the first anonymous function, s, is given the inferred type string, and the expression TimeSpan. Parse(s) is related to the return type of f1, inferring Y to be System.TimeSpan. Finally, the parameter of the second anonymous function, t, is given the inferred type System.TimeSpan, and the expression t.TotalSeconds is related to the return type of f2, inferring Z to be double. Thus, the result of the invocation is of type double. *end example*]

# 12.6.3.14 Type inference for conversion of method groups

Similar to calls of generic methods, type inference shall also be applied when a method group M containing a generic method is converted to a given delegate type D (§11.8). Given a method

 $T_r M < X_1 ... X_n > (T_1 X_1 ... T_m X_m)$ 

and the method group M being assigned to the delegate type D the task of type inference is to find type arguments  $S_{1...}S_n$  so that the expression:

M<S1....Sn>

becomes compatible (§20.2) with D.

Unlike the type inference algorithm for generic method calls, in this case, there are only argument *types*, no argument *expressions*. In particular, there are no anonymous functions and hence no need for multiple phases of inference.

Instead, all  $X_i$  are considered *unfixed*, and a *lower-bound inference* is made *from* each argument type  $U_j$  of D to the corresponding parameter type  $T_j$  of M. If for any of the  $X_i$  no bounds were found, type inference fails. Otherwise, all  $X_i$  are *fixed* to corresponding  $S_i$ , which are the result of type inference.

12.6.3.15 Finding the best common type of a set of expressions

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous functions with *block* bodies are found in this way.

The best common type for a set of expressions  $E_{1...E_m}$  is determined as follows:

• A new *unfixed* type variable X is introduced.

- For each expression Ei an *output type inference* (§12.6.3.7) is performed from it to X.
- X is *fixed* (§12.6.3.12), if possible, and the resulting type is the best common type.
- Otherwise inference fails.

[Note: Intuitively this inference is equivalent to calling a method

void M<X>(X  $x_1 \dots X x_m$ )

with the Ei as arguments and inferring X. end note]

# 12.6.4 Overload resolution

# 12.6.4.1 General

Overload resolution is a binding-time mechanism for selecting the best function member to invoke given an argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an invocation-expression (§12.7.6).
- Invocation of an instance constructor named in an object-creation-expression (§12.7.11.2).
- Invocation of an indexer accessor through an *element-access* (§12.7.7).
- Invocation of a predefined or user-defined operator referenced in an expression (§12.4.4 and §12.4.5).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way. For instance, the set of candidates for a method invocation does not include methods marked override (§12.5), and methods in a base class are not candidates if any method in a derived class is applicable (§12.7.6.2).

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- First, the set of candidate function members is reduced to those function members that are applicable with respect to the given argument list (§12.6.4.2). If this reduced set is empty, a compile-time error occurs.
- Then, the best function member from the set of applicable candidate function members is located. If the set contains only one function member, then that function member is the best function member. Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in §12.6.4.3. If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a binding-time error occurs.

The following subclauses define the exact meanings of the terms *applicable function member* and *better function member*.

# 12.6.4.2 Applicable function member

A function member is said to be an *applicable function member* with respect to an argument list A when all of the following are true:

- Each argument in A corresponds to a parameter in the function member declaration as described in §12.6.2.2, at most one argument corresponds to each parameter, and any parameter to which no argument corresponds is an optional parameter.
- For each argument in A, the parameter-passing mode of the argument is identical to the parameterpassing mode of the corresponding parameter, and
  - for a value parameter or a parameter array, an implicit conversion (§11.2) exists from the argument expression to the type of the corresponding parameter, or

• for a ref or out parameter, there is an identity conversion between the type of the argument expression and the type of the corresponding parameter

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its **normal form**. If a function member that includes a parameter array is not applicable in its normal form, the function member might instead be applicable in its **expanded form**:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list A matches the total number of parameters. If A has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in A the parameter-passing mode of the argument is identical to the parameter-passing mode of the corresponding parameter, and
  - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion (§11.2) exists from the argument expression to the type of the corresponding parameter, or
  - for a ref or out parameter, the type of the argument expression is identical to the type of the corresponding parameter.

# 12.6.4.3 Better function member

For the purposes of determining the better function member, a stripped-down argument list A is constructed containing just the argument expressions themselves in the order they appear in the original argument list.

Parameter lists for each of the candidate function members are constructed in the following way:

- The expanded form is used if the function member was applicable only in the expanded form.
- Optional parameters with no corresponding arguments are removed from the parameter list
- The parameters are reordered so that they occur at the same position as the corresponding argument in the argument list.

Given an argument list A with a set of argument expressions {  $E_1$ ,  $E_2$ , ...,  $E_N$  } and two applicable function members  $M_P$  and  $M_Q$  with parameter types {  $P_1$ ,  $P_2$ , ...,  $P_N$  } and {  $Q_1$ ,  $Q_2$ , ...,  $Q_N$  },  $M_P$  is defined to be a **better** *function member* than  $M_Q$  if

- for each argument, the implicit conversion from  $E_x$  to  $Q_x$  is not better than the implicit conversion from  $E_x$  to  $P_x$ , and
- for at least one argument, the conversion from E<sub>x</sub> to P<sub>x</sub> is better than the conversion from E<sub>x</sub> to Q<sub>x</sub>.

In case the parameter type sequences  $\{P_1, P_2, ..., P_N\}$  and  $\{Q_1, Q_2, ..., Q_N\}$  are equivalent (i.e., each  $P_i$  has an identity conversion to the corresponding  $Q_i$ ), the following tie-breaking rules are applied, in order, to determine the better function member.

- If  $M_P$  is a non-generic method and  $M_Q$  is a generic method, then  $M_P$  is better than  $M_Q$ .
- Otherwise, if  $M_P$  is applicable in its normal form and  $M_Q$  has a params array and is applicable only in its expanded form, then  $M_P$  is better than  $M_Q$ .
- Otherwise, if both methods have params arrays and are applicable only in their expanded forms, and if the params array of  $M_P$  has fewer elements than the params array of  $M_Q$ , then  $M_P$  is better than  $M_Q$ .
- Otherwise, if M<sub>P</sub> has more specific parameter types than M<sub>Q</sub>, then M<sub>P</sub> is better than M<sub>Q</sub>. Let {R<sub>1</sub>, R<sub>2</sub>, ..., R<sub>N</sub>} and {S<sub>1</sub>, S<sub>2</sub>, ..., S<sub>N</sub>} represent the uninstantiated and unexpanded parameter types of M<sub>P</sub> and M<sub>Q</sub>. M<sub>P</sub>'s parameter types are more specific than M<sub>Q</sub>'s if, for each parameter, R<sub>x</sub> is not less specific than S<sub>x</sub>, and, for at least one parameter, R<sub>x</sub> is more specific than S<sub>x</sub>:
  - A type parameter is less specific than a non-type parameter.

- Recursively, a constructed type is more specific than another constructed type (with the same number of type arguments) if at least one type argument is more specific and no type argument is less specific than the corresponding type argument in the other.
- An array type is more specific than another array type (with the same number of dimensions) if the element type of the first is more specific than the element type of the second.
- Otherwise if one member is a non-lifted operator and the other is a lifted operator, the non-lifted one is better.
- If neither function member was found to be better, and all parameters of M<sub>P</sub> have a corresponding argument whereas default arguments need to be substituted for at least one optional parameter in M<sub>Q</sub>, then M<sub>P</sub> is better than M<sub>Q</sub>. Otherwise, no function member is better.

# 12.6.4.4 Better conversion from expression

Given an implicit conversion  $C_1$  that converts from an expression E to a type  $T_1$ , and an implicit conversion  $C_2$  that converts from an expression E to a type  $T_2$ ,  $C_1$  is a **better conversion** than  $C_2$  if at least one of the following holds:

- E has a type S and an identity conversion exists from S to T<sub>1</sub> but not from S to T<sub>2</sub>
- E is not an anonymous function and  $T_1$  is a better conversion target than  $T_2$  (§12.6.4.6)
- E is an anonymous function, T<sub>1</sub> is either a delegate type D<sub>1</sub> or an expression tree type Expression<D<sub>1</sub>>, T<sub>2</sub> is either a delegate type D<sub>2</sub> or an expression tree type Expression<D<sub>2</sub>> and one of the following holds:
  - $\circ$  D<sub>1</sub> is a better conversion target than D<sub>2</sub>
  - $\circ$  D<sub>1</sub> and D<sub>2</sub> have identical parameter lists, and one of the following holds:
    - D<sub>1</sub> has a return type Y<sub>1</sub>, and D<sub>2</sub> has a return type Y<sub>2</sub>, an inferred return type X exists for E in the context of that parameter list (§12.6.3.13), and the conversion from X to Y<sub>1</sub> is better than the conversion from X to Y<sub>2</sub>
    - E is async, D<sub>1</sub> has a return type Task<Y<sub>1</sub>>, and D<sub>2</sub> has a return type Task<Y<sub>2</sub>>, an inferred return type Task<X> exists for E in the context of that parameter list (§12.6.3.13), and the conversion from X to Y<sub>1</sub> is better than the conversion from X to Y<sub>2</sub>
    - D<sub>1</sub> has a return type Y, and D<sub>2</sub> is void returning

# 12.6.4.5 Better conversion from type

Given a conversion  $C_1$  that converts from a type S to a type  $T_1$ , and a conversion  $C_2$  that converts from a type S to a type  $T_2$ ,  $C_1$  is a **better conversion** than  $C_2$  if at least one of the following holds:

- An identity conversion exists from S to T<sub>1</sub> but not from S to T<sub>2</sub>
- T<sub>1</sub> is a better conversion target than T<sub>2</sub> (§12.6.4.6)

#### 12.6.4.6 Better conversion target

Given two different types  $T_1$  and  $T_2$ ,  $T_1$  is a better conversion target than  $T_2$  if at least one of the following holds:

- An implicit conversion from T<sub>1</sub> to T<sub>2</sub> exists, and no implicit conversion from T<sub>2</sub> to T<sub>1</sub> exists
- T<sub>1</sub> is a signed integral type and T<sub>2</sub> is an unsigned integral type. Specifically:
  - $\circ \quad T_1 \text{ is sbyte and } T_2 \text{ is byte, ushort, uint, or ulong}$
  - $\circ$  T<sub>1</sub> is short and T<sub>2</sub> is ushort, uint, or ulong

- $\circ$  T<sub>1</sub> is int and T<sub>2</sub> is uint, or ulong
- $\circ$  T<sub>1</sub> is long and T<sub>2</sub> is ulong

#### 12.6.4.7 Overloading in generic classes

[*Note*: While signatures as declared shall be unique (§9.6), it is possible that substitution of type arguments results in identical signatures. In such a situation, overload resolution will pick the most specific (§12.6.4.3) of the original signatures (before substitution of type arguments), if it exists, and otherwise report an error. *end note*]

[Example: The following examples show overloads that are valid and invalid according to this rule:

```
interface I1<T> {...}
interface I2<T> {...}
class G1<U>
{
   int F1(U u);
                                   // Overload resulption for G<int>.F1
   int F1(int i);
                                   // will pick non-generic
                                   // Valid overload
   void F2(I1 < U > a);
   void F2(I2<U> a);
}
class G2<U,V>
                                   // Valid, but overload resolution for
   void F3(U u, V v);
   void F3(V v, U u);
                                   // G2<int,int>.F3 will fail
                                   // Valid, but overload resolution for
// G2<I1<int>,int>.F4 will fail
   void F4(U u, I1<V> v);
   void F4(I1\langle V \rangle v, U u);
   void F5(U u1, I1<V> v2);
void F5(V v1, U u2);
                                   // Valid overload
   void F6(ref U u);
                                   // valid overload
   void F6(out V v);
}
```

end example]

# 12.6.5 Compile-time checking of dynamic member invocation

Even though overload resolution of a dynamically bound operation takes place at run-time, it is sometimes possible at compile-time to know the list of function members from which an overload will be chosen:

- For a delegate invocation (§12.7.6.4), the list is a single function member with the same parameter list as the *delegate-type* of the invocation
- For a method invocation (§12.7.6.2) on a type, or on a value whose static type is not dynamic, the set of accessible methods in the method group is known at compile-time.
- For an object creation expression (§12.7.11.2) the set of accessible constructors in the type is known at compile-time.
- For an indexer access (§12.7.7.3) the set of accessible indexers in the receiver is known at compiletime.

In these cases a limited compile-time check is performed on each member in the known set of function members, to see if it can be known for certain never to be invoked at run-time. For each function member F a modified parameter and argument list are constructed:

• First, if F is a generic method and type arguments were provided, then those are substituted for the type parameters in the parameter list. However, if type arguments were not provided, no such substitution happens.

• Then, any parameter whose type is open (i.e., contains a type parameter; see §9.4.3) is elided, along with its corresponding parameter(s).

For F to pass the check, all of the following shall hold:

- The modified parameter list for F is applicable to the modified argument list in terms of §12.6.4.2.
- All constructed types in the modified parameter list satisfy their constraints (§9.4.5).
- If the type parameters of F were substituted in the step above, their constraints are satisfied.
- If F is a static method, the method group shall not have resulted from a *member-access* whose receiver is known at compile-time to be a variable or value.
- If F is an instance method, the method group shall not have resulted from a *member-access* whose receiver is known at compile-time to be a type.

If no candidate passes this test, a compile-time error occurs.

### 12.6.6 Function member invocation

#### 12.6.6.1 General

This subclause describes the process that takes place at run-time to invoke a particular function member. It is assumed that a binding-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are static methods, static property accessors, and user-defined operators. Static function members are always non-virtual.
- Instance function members. These are instance methods, instance constructors, instance property
  accessors, and indexer accessors. Instance function members are either non-virtual or virtual, and are
  always invoked on a particular instance. The instance is computed by an instance expression, and it
  becomes accessible within the function member as this (§12.7.8). For an instance constructor, the
  instance expression is taken to be the newly allocated object.

The run-time processing of a function member invocation consists of the following steps, where M is the function member and, if M is an instance member, E is the instance expression:

- If M is a static function member:
  - The argument list is evaluated as described in §12.6.2.
  - $\circ$  M is invoked.
- Otherwise, if the type of E is a value-type V, and M is declared or overridden in V:
  - E is evaluated. If this evaluation causes an exception, then no further steps are executed. For an instance constructor, this evaluation consists of allocating storage (typically from an execution stack) for the new object. In this case E is classified as a variable.
  - If E is not classified as a variable, then a temporary local variable of E's type is created and the value of E is assigned to that variable. E is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as this within M, but not in any other way. Thus, only when E is a true variable is it possible for the caller to observe the changes that M makes to this.
  - The argument list is evaluated as described in §12.6.2.
  - M is invoked. The variable referenced by E becomes the variable referenced by this.
- Otherwise:
  - $\circ$  E is evaluated. If this evaluation causes an exception, then no further steps are executed.

- The argument list is evaluated as described in §12.6.2.
- If the type of E is a *value-type*, a boxing conversion (§11.2.8) is performed to convert E to a *class-type*, and E is considered to be of that *class-type* in the following steps. If the *value-type* is an *enum-type*, the *class-type* is System.Enum; otherwise, it is System.ValueType.
- The value of E is checked to be valid. If the value of E is null, a
   System.NullReferenceException is thrown and no further steps are executed.
- $\circ$   $\;$  The function member implementation to invoke is determined:
  - If the binding-time type of E is an interface, the function member to invoke is the implementation of M provided by the run-time type of the instance referenced by E. This function member is determined by applying the interface mapping rules (§18.6.5) to determine the implementation of M provided by the run-time type of the instance referenced by E.
  - Otherwise, if M is a virtual function member, the function member to invoke is the implementation of M provided by the run-time type of the instance referenced by E. This function member is determined by applying the rules for determining the most derived implementation (§15.6.4) of M with respect to the run-time type of the instance referenced by E.
  - Otherwise, M is a non-virtual function member, and the function member to invoke is M itself.
- The function member implementation determined in the step above is invoked. The object referenced by E becomes the object referenced by this.

The result of the invocation of an instance constructor (§12.7.11.2) is the value created. The result of the invocation of any other function member is the value, if any, returned (§13.10.5) from its body.

#### 12.6.6.2 Invocations on boxed instances

A function member implemented in a *value-type* can be invoked through a boxed instance of that *value-type* in the following situations:

- When the function member is an override of a method inherited from type *class-type* and is invoked through an instance expression of that *class-type*. [*Note*: The *class-type* will always be one of System.Object, System.ValueType or System.Enum. *end note*]
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface-type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value-type*, and this variable becomes the variable referenced by this within the function member invocation. [*Note*: In particular, this means that when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance. *end note*]

# 12.7 Primary expressions

# 12.7.1 General

Primary expressions include the simplest forms of expressions.

primary-expression: primary-no-array-creation-expression array-creation-expression primary-no-array-creation-expression: literal simple-name parenthesized-expression member-access invocation-expression element-access this-access base-access post-increment-expression post-decrement-expression object-creation-expression delegate-creation-expression anonymous-object-creation-expression typeof-expression sizeof-expression checked-expression unchecked-expression default-value-expression anonymous-method-expression

Primary expressions are divided between *array-creation-expressions* and *primary-no-array-creation-expressions*. Treating *array-creation-expression* in this way, rather than listing it along with the other simple expression forms, enables the grammar to disallow potentially confusing code such as

object o = new int[3][1];

which would otherwise be interpreted as

object o = (new int[3])[1];

# 12.7.2 Literals

A primary-expression that consists of a literal (§7.4.5) is classified as a value.

# 12.7.3 Simple names

#### 12.7.3.1 General

A *simple-name* consists of an identifier, optionally followed by a type argument list:

simple-name: identifier type-argument-list<sub>opt</sub>

A *simple-name* is either of the form I or of the form  $I < A_1, ..., A_K >$ , where I is a single identifier and  $< A_1, ..., A_K >$  is an optional *type-argument-list*. When no *type-argument-list* is specified, consider K to be zero. The *simple-name* is evaluated and classified as follows:

- If K is zero and the *simple-name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable declaration space (§8.3) contains a local variable, parameter or constant with name I, then the *simple-name* refers to that local variable, parameter or constant and is classified as a variable or value.
- If K is zero and the *simple-name* appears within a generic method declaration but outside the *attributes* of its *method-header*, and if that declaration includes a type parameter with name I, then the *simple-name* refers to that type parameter.
- Otherwise, for each instance type T (§15.3.2), starting with the instance type of the immediately enclosing type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):

- If K is zero and the declaration of T includes a type parameter with name I, then the *simple-name* refers to that type parameter.
- Otherwise, if a member lookup (§12.5) of I in T with K type arguments produces a match:
  - If T is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of this. If a type argument list was specified, it is used in calling a generic method (§12.7.6.2).
  - Otherwise, if T is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the *block* of an instance constructor, an instance method, or an instance accessor (§12.2.1), the result is the same as a member access (§12.7.5) of the form this. I. This can only happen when K is zero.
  - Otherwise, the result is the same as a member access (§12.7.5) of the form T.I or T.I<A1, ..., AK>. In this case, it is a binding-time error for the *simple-name* to refer to an instance member.
- Otherwise, for each namespace N, starting with the namespace in which the *simple-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If K is zero and I is the name of a namespace in N, then:
    - If the location where the *simple-name* occurs is enclosed by a namespace declaration for N and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name I with a namespace or type, then the *simple-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *simple-name* refers to the namespace named I in N.
  - Otherwise, if N contains an accessible type having name I and K type parameters, then:
    - If K is zero and the location where the *simple-name* occurs is enclosed by a namespace declaration for N and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name I with a namespace or type, then the *simple-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace-or-type-name* refers to the type constructed with the given type arguments.
  - Otherwise, if the location where the *simple-name* occurs is enclosed by a namespace declaration for N:
    - If K is zero and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name I with an imported namespace or type, then the *simple-name* refers to that namespace or type.
    - Otherwise, if the namespaces imported by the *using-namespace-directives* of the namespace declaration contain exactly one type having name I and K type parameters, then the *simple-name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces imported by the *using-namespace-directives* of the namespace declaration contain more than one type having name I and K type parameters, then the *simple-name* is ambiguous and a compile-time error occurs.

[*Note*: This entire step is exactly parallel to the corresponding step in the processing of a *namespace-or-type-name* (§8.8). *end note*]

• Otherwise, the *simple-name* is undefined and a compile-time error occurs.

# 12.7.3.2 Invariant meaning in blocks

For each occurrence of a given identifier as a full *simple-name* (without a type argument list) in an expression or declarator, within the local variable declaration space (§8.3) immediately enclosing that occurrence, every other occurrence of the same identifier as a full *simple-name* in an expression or declarator shall refer to the same entity. [*Note*: This rule ensures that the meaning of a name is always the same within a given block, switch block, for-, foreach- or using-statement, or anonymous function. *end note*]

[Example: The example

```
class Test
{
    double x;
    void F(bool b) {
        x = 1.0;
        if (b) {
            int x;
            x = 1;
        }
    }
}
```

results in a compile-time error because x refers to different entities within the outer block (the extent of which includes the nested block in the if statement). In contrast, the example

```
class Test
{
    double x;
    void F(bool b) {
        if (b) {
            x = 1.0;
        }
        else {
            int x;
            x = 1;
        }
    }
}
```

is permitted because the name x is never used in the outer block. *end example*]

[*Note*: The rule of invariant meaning applies only to simple names. It is perfectly valid for the same identifier to have one meaning as a simple name and another meaning as right operand of a member access (§12.7.5). *end note*] [*Example*:

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

The example above illustrates a common pattern of using the names of fields as parameter names in an instance constructor. In the example, the simple names x and y refer to the parameters, but that does not prevent the member access expressions this.x and this.y from accessing the fields. *end example*]

# 12.7.4 Parenthesized expressions

A *parenthesized-expression* consists of an *expression* enclosed in parentheses.

```
parenthesized-expression:
( expression )
```

A *parenthesized-expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a namespace or type, a compile-time error occurs. Otherwise, the result of the *parenthesized-expression* is the result of the evaluation of the contained *expression*.

# 12.7.5 Member access

# 12.7.5.1 General

A member-access consists of a primary-expression, a predefined-type, or a qualified-alias-member, followed by a "." token, followed by an identifier, optionally followed by a type-argument-list.

member-access: primary-expression . identifier type-argument-list<sub>opt</sub> predefined-type . identifier type-argument-list<sub>opt</sub> qualified-alias-member . identifier type-argument-list<sub>opt</sub>

predefined-ty	be: one of						
bool	byte	char	decimal	double	float	int	long
object	sbyte	short	string	uint	ulong	ushort	

The *qualified-alias-member* production is defined in §14.8.

A *member-access* is either of the form E.I or of the form E.I $<A_1$ , ...,  $A_K>$ , where E is a *primary-expression*, *predefined-type* or *qualified-alias-member*, I is a single identifier, and  $<A_1$ , ...,  $A_K>$  is an optional *type-argument-list*. When no *type-argument-list* is specified, consider K to be zero.

A *member-access* with a *primary-expression* of type dynamic is dynamically bound (§12.3.3). In this case, the compiler classifies the member access as a property access of type dynamic. The rules below to determine the meaning of the *member-access* are then applied at run-time, using the run-time type instead of the compile-time type of the *primary-expression*. If this run-time classification leads to a method group, then the member access shall be the *primary-expression* of an *invocation-expression*.

The *member-access* is evaluated and classified as follows:

- If K is zero and E is a namespace and E contains a nested namespace with name I, then the result is that namespace.
- Otherwise, if E is a namespace and E contains an accessible type having name I and K type parameters, then the result is that type constructed with the given type arguments.
- If E is classified as a type, if E is not a type parameter, and if a member lookup (§12.5) of I in E with K type parameters produces a match, then E.I is evaluated and classified as follows: [*Note:* When the result of such a member lookup is a method group and K is zero, the method group can contain methods having type parameters. This allows such methods to be considered for type argument inferencing. *end note*]
  - If I identifies a type, then the result is that type constructed with any given type arguments.
  - If I identifies one or more methods, then the result is a method group with no associated instance expression.
  - If I identifies a static property, then the result is a property access with no associated instance expression.
  - If I identifies a static field:
    - If the field is readonly and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field I in E.
    - Otherwise, the result is a variable, namely the static field I in E.
  - If I identifies a static event:

- If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§15.8.1), then E.I is processed exactly as if I were a static field.
- Otherwise, the result is an event access with no associated instance expression.
- If I identifies a constant, then the result is a value, namely the value of that constant.
- If I identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
- $\circ$  Otherwise, E.I is an invalid member reference, and a compile-time error occurs.
- If E is a property access, indexer access, variable, or value, the type of which is T, and a member lookup (§12.5) of I in T with K type arguments produces a match, then E.I is evaluated and classified as follows:
  - First, if E is a property or indexer access, then the value of the property or indexer access is obtained (§12.2.2) and E is reclassified as a value.
  - If I identifies one or more methods, then the result is a method group with an associated instance expression of E.
  - If I identifies an instance property, then the result is a property access with an associated instance expression of E and an associated type that is the type of the property. If T is a class type, the associated type is picked from the first declaration or override of the property found when starting with T, and searching through its base classes.
  - If T is a *class-type* and I identifies an instance field of that *class-type*:
    - If the value of E is null, then a System.NullReferenceException is thrown.
    - Otherwise, if the field is readonly and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field I in the object referenced by E.
    - Otherwise, the result is a variable, namely the field I in the object referenced by E.
  - If T is a *struct-type* and I identifies an instance field of that *struct-type*:
    - If E is a value, or if the field is readonly and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field I in the struct instance given by E.
    - Otherwise, the result is a variable, namely the field I in the struct instance given by E.
  - If I identifies an instance event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§15.8.1), and the reference does not occur as the left-hand side of a += or -= operator, then E.I is processed exactly as if I was an instance field.
    - Otherwise, the result is an event access with an associated instance expression of E.
- Otherwise, an attempt is made to process E.I as an extension method invocation (§12.7.6.3). If this fails, E.I is an invalid member reference, and a binding-time error occurs.

# 12.7.5.2 Identical simple names and type names

In a member access of the form E.I, if E is a single identifier, and if the meaning of E as a *simple-name* (§12.7.3) is a constant, field, property, local variable, or parameter with the same type as the meaning of E as a *type-name* (§8.8.1), then both possible meanings of E are permitted. The member lookup of E.I is never ambiguous, since I shall necessarily be a member of the type E in both cases. In other words, the

rule simply permits access to the static members and nested types of E where a compile-time error would otherwise have occurred. [*Example*:

```
struct Color
   public static readonly Color White = new Color(...);
public static readonly Color Black = new Color(...);
    public Color Complement() {...}
}
class A
{
    public <u>Color</u> Color;
                                               // Field Color of type Color
    void F() {
                                               // Refs Color.Black static member
        Color = Color.Black;
        Color = Color.Complement();
                                               // Invokes Complement() on Color fld
    }
    static void G() {
    <u>Color</u> c = <u>Color</u>.White;
                                               // Refs Color.White static member
    }
}
```

Within the A class, those occurrences of the Color identifier that reference the Color type are underlined, and those that reference the Color field are not underlined. *end example*]

# 12.7.6 Invocation expressions

# 12.7.6.1 General

An invocation-expression is used to invoke a method.

invocation-expression: primary-expression ( argument-list<sub>opt</sub> )

An invocation-expression is dynamically bound (§12.3.3) if at least one of the following holds:

- The primary-expression has compile-time type dynamic.
- At least one argument of the optional *argument-list* has compile-time type dynamic.

In this case, the compiler classifies the *invocation-expression* as a value of type dynamic. The rules below to determine the meaning of the *invocation-expression* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary-expression* and arguments that have the compile-time type dynamic. If the *primary-expression* does not have compile-time type dynamic, then the method invocation undergoes a limited compile-time check as described in §12.6.5.

The *primary-expression* of an *invocation-expression* shall be a method group or a value of a *delegate-type*. If the *primary-expression* is a method group, the *invocation-expression* is a method invocation (§12.7.6.2). If the *primary-expression* is a value of a *delegate-type*, the *invocation-expression* is a delegate invocation (§12.7.6.4). If the *primary-expression* is neither a method group nor a value of a *delegate-type*, a binding-time error occurs.

The optional *argument-list* (§12.6.2) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation-expression* is classified as follows:

- If the *invocation-expression* invokes a method or delegate that returns void, the result is nothing. An expression that is classified as nothing is permitted only in the context of a *statement-expression* (§13.7) or as the body of a *lambda-expression* (§12.16). Otherwise a binding-time error occurs.
- Otherwise, the result is a value, with an associated type of the return type of the method or delegate. If the invocation is of an instance method, and the receiver is of a class type T, the associated type is

picked from the first declaration or override of the method found when starting with T and searching through its base classes.

#### 12.7.6.2 Method invocations

For a method invocation, the *primary-expression* of the *invocation-expression* shall be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The binding-time processing of a method invocation of the form M(A), where M is a method group (possibly including a *type-argument-list*), and A is an optional *argument-list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method F associated with the method group M:
  - If F is non-generic, F is a candidate when:
    - M has no type argument list, and
    - F is applicable with respect to A (§12.6.4.2).
  - If F is generic and M has no type argument list, F is a candidate when:
    - Type inference (§12.6.3) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (§9.4.5), and the parameter list of F is applicable with respect to A (§12.6.4.2)
  - If F is generic and M includes a type argument list, F is a candidate when:
    - F has the same number of method type parameters as were supplied in the type argument list, and
    - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (§9.4.5), and the parameter list of F is applicable with respect to A (§12.6.4.2).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method C.F in the set, where C is the type in which the method F is declared, all methods declared in a base type of C are removed from the set. Furthermore, if C is a class type other than object, all methods declared in an interface type are removed from the set. [*Note*: This latter rule only has an effect when the method group was the result of a member lookup on a type parameter having an effective base class other than object and a non-empty effective interface set. *end note*]
- If the resulting set of candidate methods is empty, then further processing along the following steps are abandoned, and instead an attempt is made to process the invocation as an extension method invocation (§12.7.6.3). If this fails, then no applicable methods exist, and a binding-time error occurs.
- The best method of the set of candidate methods is identified using the overload resolution rules of §12.6.4. If a single best method cannot be identified, the method invocation is ambiguous, and a binding-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- *Final validation* of the chosen best method is performed:
  - The method is validated in the context of the method group: If the best method is a static method, the method group shall have resulted from a *simple-name* or a *member-access* through a type. If the best method is an instance method, the method group shall have resulted from a *simple-name*,

a *member-access* through a variable or value, or a *base-access*. If neither of these requirements is true, a binding-time error occurs.

• If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (§9.4.5) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a binding-time error occurs.

Once a method has been selected and validated at binding-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in §12.6.6.

[*Note*: The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform type inference and overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. If no method was found, try instead to process the invocation as an extension-method invocation.*end note*]

12.7.6.3 Extension method invocations

In a method invocation (§12.6.6.2) of one of the forms

```
expr . identifier ( )
expr . identifier ( args )
expr . identifier < typeargs > ( )
expr . identifier < typeargs > ( args )
```

if the normal processing of the invocation finds no applicable methods, an attempt is made to process the construct as an extension method invocation. If *expr* or any of the *args* has compile-time type dynamic, extension methods will not apply.

The objective is to find the best *type-name* C, so that the corresponding static method invocation can take place:

C . identifier ( expr )
C . identifier ( expr , args )
C . identifier < typeargs > ( expr )
C . identifier < typeargs > ( expr , args )

An extension method Ci . Mj is *eligible* if:

- C<sub>i</sub> is a non-generic, non-nested class
- The name of M<sub>j</sub> is *identifier*
- M<sub>j</sub> is accessible and applicable when applied to the arguments as a static method as shown above
- An implicit identity, reference or boxing conversion exists from *expr* to the type of the first parameter of M<sub>j</sub>.

The search for C proceeds as follows:

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
  - If the given namespace or compilation unit directly contains non-generic type declarations C<sub>i</sub> with eligible extension methods M<sub>j</sub>, then the set of those extension methods is the candidate set.

- If namespaces imported by using namespace directives in the given namespace or compilation unit directly contain non-generic type declarations C<sub>1</sub> with eligible extension methods M<sub>j</sub>, then the set of those extension methods is the candidate set.
- If no candidate set is found in any enclosing namespace declaration or compilation unit, a compile-time error occurs.
- Otherwise, overload resolution is applied to the candidate set as described in §12.6.4. If no single best method is found, a compile-time error occurs.
- C is the type within which the best method is declared as an extension method.

Using C as a target, the method call is then processed as a static method invocation (§12.6.6). [*Note*: Unlike an instance method invocation, no exception is thrown when *expr* evaluates to a null reference. Instead, this null value is passed to the extension method as it would be via a regular static method invocation. It is up to the extension method implementation to decide how to respond to such a call. *end note*]

The preceding rules mean that instance methods take precedence over extension methods, that extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations, and that extension methods declared directly in a namespace take precedence over extension methods imported into that same namespace with a using namespace directive. [*Example*:

```
public static class E
   public static void F(this object obj, int i) { }
   public static void F(this object obj, string s) { }
}
class A { }
class B
{
   public void F(int i) { }
}
class C
{
   public void F(object obj) { }
}
class X
   static void Test(A a, B b, C c) {
                           // E.F(object, int)
// E.F(object, string)
      a.F(1);
a.F("hello");
       b.F(1);
                           // B.F(int)
      b.F("héllo");
                           // E.F(object, string)
      c.F(1);
c.F("hello");
                           // C.F(object)
                           // C.F(object)
   }
}
```

In the example, B's method takes precedence over the first extension method, and C's method takes precedence over both extension methods.

```
public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}
```

```
namespace N1
   public static class D
      public static void F(this int i) { Console.WriteLine("D.F({0})",
i); }
      public static void G(this int i) { Console.WriteLine("D.G({0})",
}
namespace N2
Ł
   using N1;
   public static class E
      public static void F(this int i) { Console.WriteLine("E.F({0})",
class Test
      static void Main(string[] args)
         1.F();
         2.G();
3.H();
      }
   }
}
```

The output of this example is:

E.F(1) D.G(2) C.H(3)

D.G takes precendece over C.G, and E.F takes precedence over both D.F and C.F. end example]

# 12.7.6.4 Delegate invocations

For a delegate invocation, the *primary-expression* of the *invocation-expression* shall be a value of a *delegate-type*. Furthermore, considering the *delegate-type* to be a function member with the same parameter list as the *delegate-type*, the *delegate-type* shall be applicable (§12.6.4.2) with respect to the *argument-list* of the *invocation-expression*.

The run-time processing of a delegate invocation of the form D(A), where D is a *primary-expression* of a *delegate-type* and A is an optional *argument-list*, consists of the following steps:

- D is evaluated. If this evaluation causes an exception, no further steps are executed.
- The argument list A is evaluated. If this evaluation causes an exception, no further steps are executed.
- The value of D is checked to be valid. If the value of D is null, a System.NullReferenceException is thrown and no further steps are executed.
- Otherwise, D is a reference to a delegate instance. Function member invocations (§12.6.6) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

See §20.6 for details of multiple invocation lists without parameters.

# 12.7.7 Element access

### 12.7.7.1 General

An *element-access* consists of a *primary-no-array-creation-expression*, followed by a "[" token, followed by an *argument-list*, followed by a "]" token. The *argument-list* consists of one or more *arguments*, separated by commas.

element-access: primary-no-array-creation-expression [ argument-list ]

The *argument-list* of an *element-access* is not allowed to contain ref or out arguments.

An *element-access* is dynamically bound (§12.3.3) if at least one of the following holds:

- The primary-no-array-creation-expression has compile-time type dynamic.
- At least one expression of the *argument-list* has compile-time type dynamic and the *primary-no-array-creation-expression* does not have an array type.

In this case, the compiler classifies the *element-access* as a value of type dynamic. The rules below to determine the meaning of the *element-access* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary-no-array-creation-expression* and *argument-list* expressions which have the compile-time type dynamic. If the *primary-no-array-creation-expression* does not have compile-time type dynamic, then the element access undergoes a limited compile-time check as described in §12.6.5.

If the *primary-no-array-creation-expression* of an *element-access* is a value of an *array-type*, the *element-access* is an array access (§12.7.7.2). Otherwise, the *primary-no-array-creation-expression* shall be a variable or value of a class, struct, or interface type that has one or more indexer members, in which case the *element-access* is an indexer access (§12.7.7.3).

#### 12.7.7.2 Array access

For an array access, the *primary-no-array-creation-expression* of the *element-access* shall be a value of an *array-type*. Furthermore, the *argument-list* of an array access is not allowed to contain named arguments. The number of expressions in the *argument-list* shall be the same as the rank of the *array-type*, and each expression shall be of type int, uint, long, ulong, or shall be implicitly convertible to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value(s) of the expression(s) in the *argument-list*.

The run-time processing of an array access of the form P[A], where P is a *primary-no-array-creation-expression* of an *array-type* and A is an *argument-list*, consists of the following steps:

- P is evaluated. If this evaluation causes an exception, no further steps are executed.
- The index expressions of the *argument-list* are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion (§11.2) to one of the following types is performed: int, uint, long, ulong. The first type in this list for which an implicit conversion exists is chosen. For instance, if the index expression is of type short then an implicit conversion to int is performed, since implicit conversions from short to int and from short to long are possible. If evaluation of an index expression or the subsequent implicit conversion causes an exception, then no further index expressions are evaluated and no further steps are executed.
- The value of P is checked to be valid. If the value of P is null, a System.NullReferenceException is thrown and no further steps are executed.
- The value of each expression in the *argument-list* is checked against the actual bounds of each dimension of the array instance referenced by P. If one or more values are out of range, a System.IndexOutOfRangeException is thrown and no further steps are executed.

• The location of the array element given by the index expression(s) is computed, and this location becomes the result of the array access.

### 12.7.7.3 Indexer access

For an indexer access, the *primary-no-array-creation-expression* of the *element-access* shall be a variable or value of a class, struct, or interface type, and this type shall implement one or more indexers that are applicable with respect to the *argument-list* of the *element-access*.

The binding-time processing of an indexer access of the form P[A], where P is a *primary-no-array-creation-expression* of a class, struct, or interface type T, and A is an *argument-list*, consists of the following steps:

- The set of indexers provided by T is constructed. The set consists of all indexers declared in T or a base type of T that are not override declarations and are accessible in the current context (§8.5).
- The set is reduced to those indexers that are applicable and not hidden by other indexers. The following rules are applied to each indexer S.I in the set, where S is the type in which the indexer I is declared:
  - If I is not applicable with respect to A (§12.6.4.2), then I is removed from the set.
  - If I is applicable with respect to A (§12.6.4.2), then all indexers declared in a base type of S are removed from the set.
  - If I is applicable with respect to A (§12.6.4.2) and S is a class type other than object, all indexers declared in an interface are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a binding-time error occurs.
- The best indexer of the set of candidate indexers is identified using the overload resolution rules of §12.6.4. If a single best indexer cannot be identified, the indexer access is ambiguous, and a binding-time error occurs.
- The index expressions of the *argument-list* are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of P and an associated argument list of A, and an associated type that is the type of the indexer. If T is a class type, the associated type is picked from the first declaration or override of the indexer found when starting with T and searching through its base classes.

Depending on the context in which it is used, an indexer access causes invocation of either the *get-accessor* or the *set-accessor* of the indexer. If the indexer access is the target of an assignment, the *set-accessor* is invoked to assign a new value (§12.18.2). In all other cases, the *get-accessor* is invoked to obtain the current value (§12.2.2).

# 12.7.8 This access

A *this-access* consists of the keyword this.

this-access: this

A *this-access* is permitted only in the *block* of an instance constructor, an instance method, an instance accessor (§12.2.1), or a finalizer. It has one of the following meanings:

- When this is used in a *primary-expression* within an instance constructor of a class, it is classified as a value. The type of the value is the instance type (§15.3.2) of the class within which the usage occurs, and the value is a reference to the object being constructed.
- When this is used in a *primary-expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the instance type (§15.3.2) of the class within which

the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.

- When this is used in a *primary-expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the instance type (§15.3.2) of the struct within which the usage occurs, and the variable represents the struct being constructed.
  - If the constructor declaration has no constructor initializer, the this variable behaves exactly the same as an out parameter of the struct type. In particular, this means that the variable shall be definitely assigned in every execution path of the instance constructor.
  - Otherwise, the this variable behaves exactly the same as a ref parameter of the struct type. In particular, this means that the variable is considered initially assigned.
- When this is used in a *primary-expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the instance type (§15.3.2) of the struct within which the usage occurs.
  - If the method or accessor is not an iterator (§15.14) or async function (§15.15), the this variable represents the struct for which the method or accessor was invoked, and behaves exactly the same as a ref parameter of the struct type.
  - If the method or accessor is an iterator or async function, the this variable represents a *copy* of the struct for which the method or accessor was invoked, and behaves exactly the same as a *value* parameter of the struct type.

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.

# 12.7.9 Base access

A *base-access* consists of the keyword base followed by either a "." token and an identifier and optional *type-argument-list* or an *argument-list* enclosed in square brackets:

base-access: base . identifier type-argument-list<sub>opt</sub> base [ argument-list ]

A *base-access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base-access* is permitted only in the *block* of an instance constructor, an instance method, an instance accessor (§12.2.1), or a finalizer. When base.I occurs in a class or struct, I shall denote a member of the base class of that class or struct. Likewise, when base[E] occurs in a class, an applicable indexer shall exist in the base class.

At binding-time, *base-access* expressions of the form base.I and base[E] are evaluated exactly as if they were written ((B)this).I and ((B)this)[E], where B is the base class of the class or struct in which the construct occurs. Thus, base.I and base[E] correspond to this.I and this[E], except this is viewed as an instance of the base class.

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 (§12.6.6) is changed. The function member that is invoked is determined by finding the most derived implementation (§15.6.4) 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 binding-time error occurs.

[Note: Unlike this, base is not an expression in itself. It is a keyword only used in the context of a baseaccess or a constructor-initializer (§15.11.2). end note]

#### 12.7.10 Postfix increment and decrement operators

post-increment-expression: primary-expression ++ post-decrement-expression: primary-expression --

The operand of a postfix increment or decrement operation shall be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the *primary-expression* has the compile-time type dynamic then the operator is dynamically bound (§12.3.3), the *post-increment-expression* or *post-decrement-expression* has the compile-time type dynamic and the following rules are applied at run-time using the run-time type of the *primary-expression*.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer shall have both a get and a set accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (§12.4.4) is applied to select a specific operator implementation. Predefined ++ and -- operators exist for the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any enum type. The predefined ++ operators return the value produced by adding 1 to the operand, and the predefined -- operators return the value produced by subtracting 1 from the operand. In a checked context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a System.OverflowException is thrown.

There shall be an implicit conversion from the return type of the selected unary operator to the type of the *primary-expression*, otherwise a compile-time error occurs.

The run-time processing of a postfix increment or decrement operation of the form x++ or x-- consists of the following steps:

- If x is classified as a variable:
  - x is evaluated to produce the variable.
  - The value of x is saved.
  - The saved value of x is converted to the operand type of the selected operator and the operator is invoked with this value as its argument.
  - The value returned by the operator is converted to the type of x and stored in the location given by the earlier evaluation of x.
  - The saved value of x becomes the result of the operation.
- If x is classified as a property or indexer access:
  - The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent get and set accessor invocations.
  - $\circ$  The get accessor of x is invoked and the returned value is saved.
  - The saved value of x is converted to the operand type of the selected operator and the operator is invoked with this value as its argument.
  - The value returned by the operator is converted to the type of x and the set accessor of x is invoked with this value as its value argument.
  - The saved value of x becomes the result of the operation.

The ++ and -- operators also support prefix notation (§12.8.6). Typically, the result of x++ or x-- is the value of x *before* the operation, whereas the result of ++x or --x is the value of x *after* the operation. In either case, x itself has the same value after the operation.

An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

### 12.7.11 The new operator

#### 12.7.11.1 General

The new operator is used to create new instances of types.

There are three forms of new expressions:

- Object creation expressions and anonymous object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to obtain instances of delegate types.

The new operator implies creation of an instance of a type, but does not necessarily imply allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no allocations occur when new is used to create instances of value types.

[*Note*: Delegate creation expressions do not always create new instances. When the expression is processed in the same way as a method group conversion (§11.8) or an anonymous function conversion (§11.7) this may result in an existing delegate instance being reused. *end note*]

#### 12.7.11.2 Object creation expressions

An object-creation-expression is used to create a new instance of a class-type or a value-type.

object-creation-expression: new type ( argument-list<sub>opt</sub> ) object-or-collection-initializer new type object-or-collection-initializer object-or-collection-initializer: object-initializer

collection-initializer

The type of an object-creation-expression shall be a class-type, a value-type, or a type-parameter. The type cannot be an abstract or static class-type.

The optional *argument-list* (§12.6.2) is permitted only if the *type* is a *class-type* or a *struct-type*.

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Processing of an object creation expression that includes an object initializer or collection initializer consists of first processing the instance constructor and then processing the member or element initializations specified by the object initializer (§12.7.11.3) or collection initializer (§12.7.11.4).

If any of the arguments in the optional argument-list has the compile-time type dynamic then the *object-creation-expression* is dynamically bound (§12.3.3) and the following rules are applied at run-time using the run-time type of those arguments of the *argument-list* that have the compile-time type dynamic. However, the object creation undergoes a limited compile-time check as described in §12.6.5.

The binding-time processing of an *object-creation-expression* of the form new T(A), where T is a *class-type*, or a *value-type*, and A is an optional *argument-list*, consists of the following steps:

- If T is a *value-type* and A is not present:
  - The *object-creation-expression* is a default constructor invocation. The result of the *object-creation-expression* is a value of type T, namely the default value for T as defined in §9.3.3.
- Otherwise, if T is a *type-parameter* and A is not present:

- If no value type constraint or constructor constraint (§15.2.5) has been specified for T, a bindingtime error occurs.
- The result of the *object-creation-expression* is a value of the run-time type that the type parameter has been bound to, namely the result of invoking the default constructor of that type. The run-time type may be a reference type or a value type.
- Otherwise, if T is a *class-type* or a *struct-type*:
  - If T is an abstract or static *class-type*, a compile-time error occurs.
  - The instance constructor to invoke is determined using the overload resolution rules of §12.6.4.
     The set of candidate instance constructors consists of all accessible instance constructors declared in T, which are applicable with respect to A (§12.6.4.2). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a binding-time error occurs.
  - The result of the *object-creation-expression* is a value of type T, namely the value produced by invoking the instance constructor determined in the step above.
  - Otherwise, the *object-creation-expression* is invalid, and a binding-time error occurs.

Even if the *object-creation-expression* is dynamically bound, the compile-time type is still T.

The run-time processing of an *object-creation-expression* of the form new T(A), where T is *class-type* or a *struct-type* and A is an optional *argument-list*, consists of the following steps:

- If T is a *class-type*:
  - A new instance of class T is allocated. If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown and no further steps are executed.
  - All fields of the new instance are initialized to their default values (§10.3).
  - The instance constructor is invoked according to the rules of function member invocation (§12.6.6). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as this.
- If T is a *struct-type*:
  - An instance of type T is created by allocating a temporary local variable. Since an instance constructor of a *struct-type* is required to definitely assign a value to each field of the instance being created, no initialization of the temporary variable is necessary.
  - The instance constructor is invoked according to the rules of function member invocation (§12.6.6). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as this.

# 12.7.11.3 Object initializers

An *object initializer* specifies values for zero or more fields or properties of an object.

```
object-initializer:

{ member-initializer-list<sub>opt</sub> }

{ member-initializer-list , }

member-initializer

member-initializer

member-initializer-list , member-initializer

member-initializer:

identifier = initializer-value

initializer-value:

expression

object-or-collection-initializer
```

An object initializer consists of a sequence of member initializers, enclosed by { and } tokens and separated by commas. Each member initializer shall name an accessible field or property of the object being initialized, followed by an equals sign and an expression or an object initializer or collection initializer. It is an error for an object initializer to include more than one member initializer for the same field or property. It is not possible for the object initializer to refer to the newly created object it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment (§12.18.2) to the field or property.

A member initializer that specifies an object initializer after the equals sign is a **nested object initializer**, i.e., an initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the nested object initializer are treated as assignments to members of the field or property. Nested object initializers cannot be applied to properties with a value type, or to read-only fields with a value type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the field or property, the elements given in the initializer are added to the collection referenced by the field or property. The field or property shall be of a collection type that satisfies the requirements specified in §12.7.11.4.

[Example: The following class represents a point with two coordinates:

```
public class Point
{
    int x, y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

An instance of Point can be created and initialized as follows:

Point a = new Point { X = 0, Y = 1 };

which has the same effect as

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

where <u>a</u> is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

```
public class Rectangle
{
    Point p1, p2;
    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

An instance of Rectangle can be created and initialized as follows:

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

where \_\_\_r, \_\_p1 and \_\_\_p2 are temporary variables that are otherwise invisible and inaccessible.

If Rectangle's constructor allocates the two embedded Point instances

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();
    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

the following construct can be used to initialize the embedded Point instances instead of assigning new instances:

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

end example]

12.7.11.4 Collection initializers

A collection initializer specifies the elements of a collection.

```
collection-initializer:
{ element-initializer-list }
{ element-initializer-list , }
element-initializer-list:
element-initializer
```

```
element-initializer-list , element-initializer element-initializer:
```

```
non-assignment-expression
{ expression-list }
```

```
expression-list:
expression
expression-list , expression
```

A collection initializer consists of a sequence of element initializers, enclosed by { and } tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by { and } tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment

expression, to avoid ambiguity with member initializers. The *non-assignment-expression* production is defined in §12.19.

### [Example:

The following is an example of an object creation expression that includes a collection initializer:

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

#### end example]

The collection object to which a collection initializer is applied shall be of a type that implements System.Collections.IEnumerable or a compile-time error occurs. For each specified element in order, the collection initializer invokes an Add method on the target object with the expression list of the element initializer as argument list, applying normal overload resolution for each invocation. Thus, the collection object shall contain an applicable Add method for each element initializer.

[Example: The following class represents a contact with a name and a list of phone numbers:

```
public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();
    public string Name { get { return name; } set { name = value; } }
    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

A List<Contact> can be created and initialized as follows:

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

which has the same effect as

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

where \_\_clist, \_\_c1 and \_\_c2 are temporary variables that are otherwise invisible and inaccessible. *end example*]

#### 12.7.11.5 Array creation expressions

An *array-creation-expression* is used to create a new instance of an *array-type*.

array-creation-expression:

```
new non-array-type [ expression-list ] rank-specifiers<sub>opt</sub> array-initializer<sub>opt</sub>
new array-type array-initializer
new rank-specifier array-initializer
```

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. [*Example*: The array creation expression new int[10,20] produces an array instance of type int[,], and the array creation expression new int[10][,] produces an array instance of type int[][,]. *end example*] Each expression in the expression list shall be of type int, uint, long, or ulong, or implicitly convertible to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension shall be nonnegative, it is a compile-time error to have a constant expression with a negative value, in the expression list.

Except in an unsafe context (§23.2), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list shall be a constant and the rank and dimension lengths specified by the expression list shall match those of the array initializer.

In an array creation expression of the second or third form, the rank of the specified array type or rank specifier shall match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

new int[,] {{0, 1}, {2, 3}, {4, 5}}

exactly corresponds to

new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}

An array creation expression of the third form is referred to as an *implicitly typed array-creation expression*. It is similar to the second form, except that the element type of the array is not explicitly given, but determined as the best common type (§12.6.3.15) of the set of expressions in the array initializer. For a multidimensional array, i.e., one where the *rank-specifier* contains at least one comma, this set comprises all *expressions* found in nested *array-initializers*.

Array initializers are described further in §17.7.

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The run-time processing of an array creation expression consists of the following steps:

- The dimension length expressions of the *expression-list* are evaluated in order, from left to right.
   Following evaluation of each expression, an implicit conversion (§11.2) to one of the following types is performed: int, uint, long, ulong. The first type in this list for which an implicit conversion exists is chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated, as follows: If one or more of the values are less than zero, a System.OverflowException is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values (§10.3).
- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array shall be manually initialized. [*Example*: The statement

int[][] a = new int[100][];

creates a single-dimensional array with 100 elements of type int[]. The initial value of each element is null. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

int[][] a = new int[100][5]; // Error

results in a compile-time error. Instantiation of the sub-arrays can instead be performed manually, as in

int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];</pre>

end example]

[*Note*: When an array of arrays has a "rectangular" shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects—one outer array and 100 sub-arrays. In contrast,

int[,] = new int[100, 5];

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement. *end note*]

[*Example*: The following are examples of implicitly typed array creation expressions:

The last expression causes a compile-time error because neither int nor string is implicitly convertible to the other, and so there is no best common type. An explicitly typed array creation expression must be used in this case, for example specifying the type to be object[]. Alternatively, one of the elements can be cast to a common base type, which would then become the inferred element type. *end example*]

Implicitly typed array creation expressions can be combined with anonymous object initializers (§12.7.11.7) to create anonymously typed data structures. [*Example*:

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

end example]

12.7.11.6 Delegate creation expressions

A *delegate-creation-expression* is used to obtain an instance of a *delegate-type*.

delegate-creation-expression: new delegate-type ( expression )

The argument of a delegate creation expression shall be a method group, an anonymous function, or a value of either the compile-time type dynamic or a *delegate-type*. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is an anonymous function it directly defines the parameters and method body of the delegate target. If the argument is a value it identifies a delegate instance of which to create a copy.

If the *expression* has the compile-time type dynamic, the *delegate-creation-expression* is dynamically bound (§12.7.11.6), and the rules below are applied at run-time using the run-time type of the *expression*. Otherwise, the rules are applied at compile-time.

The binding-time processing of a *delegate-creation-expression* of the form new D(E), where D is a *delegate-type* and E is an *expression*, consists of the following steps:

- If E is a method group, the delegate creation expression is processed in the same way as a method group conversion (§11.8) from E to D.
- If E is an anonymous function, the delegate creation expression is processed in the same way as an anonymous function conversion (§11.7) from E to D.
- If E is a value, E shall be compatible (§20.2) with D, and the result is a reference to a newly created delegate with a single-entry invocation list that invokes E.

The run-time processing of a *delegate-creation-expression* of the form new D(E), where D is a *delegate-type* and E is an *expression*, consists of the following steps:

- If E is a method group, the delegate creation expression is evaluated as a method group conversion (§11.8) from E to D.
- If E is an anonymous function, the delegate creation is evaluated as an anonymous function conversion from E to D (§11.7).
- If E is a value of a *delegate-type*:
  - E is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the value of E is null, a System.NullReferenceException is thrown and no further steps are executed.
  - A new instance of the delegate type D is allocated. If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown and no further steps are executed.
  - The new delegate instance is initialized with a single-entry invocation list that invokes E.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. [*Note*: Remember, when two delegates are combined or one is removed from another, a new delegate results; no existing delegate has its content changed. *end note*]

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, finalizer, or static constructor.

[*Example*: As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);
class A
{
    DoubleFunc f = new DoubleFunc(Square);
    static float Square(float x) {
        return x * x;
    }
    static double Square(double x) {
        return x * x;
    }
}
```

the A.f field is initialized with a delegate that refers to the second Square method because that method exactly matches the formal parameter list and return type of DoubleFunc. Had the second Square method not been present, a compile-time error would have occurred. *end example*]

#### 12.7.11.7 Anonymous object creation expressions

An anonymous-object-creation-expression is used to create an object of an anonymous type.

anonymous-object-creation-expression: new anonymous-object-initializer

```
anonymous-object-initializer:

{ member-declarator-list<sub>opt</sub> }

{ member-declarator-list , }

member-declarator-list:

member-declarator

member-declarator-list , member-declarator

member-declarator:

simple-name

member-access

base-access

identifier = expression
```

An anonymous object initializer declares an anonymous type and returns an instance of that type. An anonymous type is a nameless class type that inherits directly from object. The members of an anonymous type are a sequence of read-only properties inferred from the anonymous object initializer used to create an instance of the type. Specifically, an anonymous object initializer of the form

new {  $p_1 = e_1$  ,  $p_2 = e_2$  , ...  $p_n = e_n$  }

declares an anonymous type of the form

```
class __Anonymous1
{
    private readonly T_1 f_1;
    private readonly T_2 f_2;
    ...
    private readonly T_n f_n;
    public __Anonymous1(T_1 a_1, T_2 a_2, ..., T_n a_n) {
        f_1 = a_1;
        f_2 = a_2;
        ...
        f_n = a_n;
    }
    public T_1 p_1 { get { return f_1; } }
    public T_2 p_2 { get { return f_2; } }
    ...
    public T_n p_n { get { return f_n; } }
    public override bool Equals(object ___0) { ... }
    public override int GetHashCode() { ... }
}
```

where each  $T_x$  is the type of the corresponding expression  $e_x$ . The expression used in a *member-declarator* shall have a type. Thus, it is a compile-time error for an expression in a *member-declarator* to be null or an anonymous function. It is also a compile-time error for the expression to have an unsafe type.

The names of an anonymous type and of the parameter to its Equals method are automatically generated by the compiler and cannot be referenced in program text.

Within the same program, two anonymous object initializers that specify a sequence of properties of the same names and compile-time types in the same order will produce instances of the same anonymous type.

[*Example*: In the example

var p1 = new { Name = "Lawnmower", Price = 495.00 }; var p2 = new { Name = "Shovel", Price = 26.95 }; p1 = p2;

the assignment on the last line is permitted because p1 and p2 are of the same anonymous type. *end example*]

The Equals and GetHashcode methods on anonymous types override the methods inherited from object, and are defined in terms of the Equals and GetHashcode of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name (§12.7.3), a member access (§12.7.5) or a base access (§12.7.9). This is called a *projection initializer* and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

identifier

expr . identifier

are precisely equivalent to the following, respectively:

```
identifier = identifier identifier = expr . identifier
```

Thus, in a projection initializer the *identifier* selects both the value and the field or property to which the value is assigned. Intuitively, a projection initializer projects not just a value, but also the name of the value.

#### 12.7.12 The typeof operator

The typeof operator is used to obtain the System. Type object for a type.

```
typeof-expression:
    typeof ( type )
    typeof ( unbound-type-name )
    typeof ( void )
unbound-type-name:
    identifier generic-dimension-specifier<sub>opt</sub>
    identifier :: identifier generic-dimension-specifier<sub>opt</sub>
    unbound-type-name . identifier generic-dimension-specifier<sub>opt</sub>
```

generic-dimension-specifier:

< commas<sub>opt</sub> >

commas:

commas ,

The first form of *typeof-expression* consists of a typeof keyword followed by a parenthesized *type*. The result of an expression of this form is the System.Type object for the indicated type. There is only one System.Type object for any given type. This means that for a type T, typeof(T) == typeof(T) is always true. The *type* cannot be dynamic.

The second form of *typeof-expression* consists of a typeof keyword followed by a parenthesized *unbound-type-name*. [*Note*: An *unbound-type-name* is very similar to a *type-name* (§8.8) except that an *unbound-type-name* contains *generic-dimension-specifiers* where a *type-name* contains *type-argument-lists*. *end note*] When the operand of a *typeof-expression* is a sequence of tokens that satisfies the grammars of both *unbound-type-name* and *type-name*, namely when it contains neither a *generic-dimension-specifier* nor a

*type-argument-list,* the sequence of tokens is considered to be a *type-name*. The meaning of an *unbound-type-name* is determined as follows:

- Convert the sequence of tokens to a *type-name* by replacing each *generic-dimension-specifier* with a *type-argument-list* having the same number of commas and the keyword object as each *type-argument*.
- Evaluate the resulting type-name, while ignoring all type parameter constraints.
- The *unbound-type-name* resolves to the *unbound generic type* associated with the resulting constructed type (§9.4).

The result of the *typeof-expression* is the System. Type object for the resulting unbound generic type.

The third form of *typeof-expression* consists of a typeof keyword followed by a parenthesized void keyword. The result of an expression of this form is the System.Type object that represents the absence of a type. The type object returned by typeof(void) is distinct from the type object returned for any type. [*Note*: This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of System.Type. *end note*]

The typeof operator can be used on a type parameter. The result is the System.Type object for the runtime type that was bound to the type parameter. The typeof operator can also be used on a constructed type or an unbound generic type (§9.4.4). The System.Type object for an unbound generic type is not the same as the System.Type object of the instance type (§15.3.2). The instance type is always a closed constructed type at run-time so its System.Type object depends on the run-time type arguments in use. The unbound generic type, on the other hand, has no type arguments, and yields the same System.Type object regardless of runtime type arguments.

#### [Example: The example

```
using System;
class X<T>
{
    public static void PrintTypes() {
   Type[] t = {
             typeof(int),
typeof(System.Int32),
             typeof(system:inc
typeof(string),
typeof(double[]),
typeof(void),
typeof(T),
             typeof(X<T>),
             typeof(X<X<T>>),
             typeof(X<>)
        };
for (int i = 0; i < t.Length; i++) {</pre>
             Console.WriteLine(t[i]);
        }
    }
}
class Test
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

produces the following output:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X 1[System.Int32]
X 1[X 1[System.Int32]]
X 1[T]
```

Note that int and System. Int32 are the same type.

The result of typeof(X<>) does not depend on the type argument but the result of typeof(X<T>) does. *end example*]

## 12.7.13 The sizeof operator

The sizeof operator returns the number of 8-bit bytes occupied by a variable of a given type. The type specified as an operand to sizeof shall be an unmanaged-type (§23.3).

For certain predefined types the sizeof operator yields a constant int value as shown in the table below:

Expression	Result
<pre>sizeof(sbyte)</pre>	1
sizeof(byte)	1
<pre>sizeof(short)</pre>	2
<pre>sizeof(ushort)</pre>	2
sizeof(int)	4
sizeof(uint)	4
<pre>sizeof(long)</pre>	8
<pre>sizeof(ulong)</pre>	8
sizeof(char)	2
sizeof(float)	4
<pre>sizeof(double)</pre>	8
<pre>sizeof(bool)</pre>	1
<pre>sizeof(decimal)</pre>	16

For an enum type T, the result of the expression sizeof(T) is a constant value equal to the size of its underlying type, as given above. For all other operand types, the sizeof operator is specified in §23.6.9.

## 12.7.14 The checked and unchecked operators

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

```
checked-expression:
    checked ( expression )
unchecked-expression:
    unchecked ( expression )
```

The checked operator evaluates the contained expression in a checked context, and the unchecked operator evaluates the contained expression in an unchecked context. A *checked-expression* or *unchecked-expression* corresponds exactly to a *parenthesized-expression* (§12.7.4), except that the contained expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the checked and unchecked statements (§13.12).

The following operations are affected by the overflow checking context established by the checked and unchecked operators and statements:

- The predefined ++ and -- operators (§12.7.10 and §12.8.6), when the operand is of an integral or enumtype.
- The predefined unary operator (§12.8.3), when the operand is of an integral type.
- The predefined +, -, \*, and / binary operators (§12.9), when both operands are of integral or enumtypes.
- Explicit numeric conversions (§11.3.2) from one integral or enumtype to another integral or enumtype, or from float or double to an integral or enumtype.

When one of the above operations produces a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a checked context, if the operation is a constant expression (§12.20), a compile-time error occurs. Otherwise, when the operation is performed at run-time, a System.OverflowException is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For non-constant expressions (§12.20) (expressions that are evaluated at run-time) that are not enclosed by any checked or unchecked operators or statements, the default overflow checking context is unchecked, unless external factors (such as compiler switches and execution environment configuration) call for checked evaluation.

For constant expressions (§12.20) (expressions that can be fully evaluated at compile-time), the default overflow checking context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

The body of an anonymous function is not affected by checked or unchecked contexts in which the anonymous function occurs.

[Example: In the following code

```
class Test
{
  static readonly int x = 1000000;
  static readonly int y = 1000000;
  static int F() {
    return checked(x * y); // Throws OverflowException
  }
  static int G() {
    return unchecked(x * y); // Returns -727379968
  }
  static int H() {
    return x * y; // Depends on default
  }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the F method throws a System.OverflowException, and the G method returns –727379968 (the lower 32 bits of the out-of-range result). The behavior of the H method depends on the default overflow-checking context for the compilation, but it is either the same as F or the same as G. *end example*]

[Example: In the following code

```
class Test
Ł
   const int x = 1000000;
   const int y = 1000000;
   static int F() {
      return checked(x * y);
                                 // Compile-time error, overflow
   static int G() {
      return unchecked(x * y);
                                 // Returns -727379968
   }
   static int H() {
      return x * y;
                                 // Compile-time error, overflow
   }
}
```

the overflows that occur when evaluating the constant expressions in F and H cause compile-time errors to be reported because the expressions are evaluated in a checked context. An overflow also occurs when evaluating the constant expression in G, but since the evaluation takes place in an unchecked context, the overflow is not reported. *end example*]

The checked and unchecked operators only affect the overflow checking context for those operations that are textually contained within the "(" and ")" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. [*Example*: In the following code

```
class Test
{
   static int Multiply(int x, int y) {
      return x * y;
   }
   static int F() {
      return checked(Multiply(1000000, 1000000));
   }
}
```

the use of checked in F does not affect the evaluation of x \* y in Multiply, so x \* y is evaluated in the default overflow checking context. *end example*]

The unchecked operator is convenient when writing constants of the signed integral types in hexadecimal notation. [*Example*:

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFF);
    public const int HighBit = unchecked((int)0x8000000);
}
```

Both of the hexadecimal constants above are of type uint. Because the constants are outside the int range, without the unchecked operator, the casts to int would produce compile-time errors. *end example*]

[*Note*: The checked and unchecked operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow *even* within an explicitly unchecked construct. Similarly, multiplying two floats never results in an exception on overflow *even* within an explicitly checked construct. In addition, other operators are *never* affected by the mode of checking, whether default or explicit. *end note*]

## 12.7.15 Default value expressions

A default value expression is used to obtain the default value (§10.3) of a type. Typically a default value expression is used for type parameters, since it might not be known if the type parameter is a value type or

a reference type. (No conversion exists from the null literal (§7.4.5.7) to a type parameter unless the type parameter is known to be a reference type (§9.2).)

default-value-expression:
 default ( type )

If the *type* in a *default-value-expression* evaluates at run-time to a reference type, the result is null converted to that type. If the *type* in a *default-value-expression* evaluates at run-time to a value type, the result is the *value-type*'s default value (§9.3.3).

A *default-value-expression* is a constant expression (§12.20) if *type* is a reference type or a type parameter that is known to be a reference type (§9.2). In addition, a *default-value-expression* is a constant expression if the type is one of the following value types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, or any enumeration type.

## 12.7.16 Anonymous method expressions

An *anonymous-method-expression* is one of two ways of defining an anonymous function. These are further described in §12.16.

## 12.8 Unary operators

## 12.8.1 General

The +, -, !, ~, ++, --, cast, and await operators are called the unary operators.

unary-expression:

primary-expression

- + unary-expression
- unary-expression
- ! unary-expression ~ unary-expression

pre-increment-expression pre-decrement-expression cast-expression await-expression

If the operand of a *unary-expression* has the compile-time type dynamic, it is dynamically bound (§12.3.3). In this case, the compile-time type of the *unary-expression* is dynamic, and the resolution described below will take place at run-time using the run-time type of the operand.

## 12.8.2 Unary plus operator

For an operation of the form +x, unary operator overload resolution (§12.4.4) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

For each of these operators, the result is simply the value of the operand.

Lifted (§12.4.8) forms of the unlifted predefined unary plus operators defined above are also predefined.

## 12.8.3 Unary minus operator

For an operation of the form -x, unary operator overload resolution (§12.4.4) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary minus operators are:

• Integer negation:

int operator -(int x); long operator -(long x);

The result is computed by subtracting x from zero. If the value of x is the smallest representable value of the operand type  $(-2^{31} \text{ for int or } -2^{63} \text{ for long})$ , then the mathematical negation of x is not representable within the operand type. If this occurs within a checked context, a System.OverflowException is thrown; if it occurs within an unchecked context, the result is the value of the operand and the overflow is not reported.

If the operand of the negation operator is of type uint, it is converted to type long, and the type of the result is long. An exception is the rule that permits the int value -2147483648 ( $-2^{31}$ ) to be written as a decimal integer literal (§7.4.5.3).

If the operand of the negation operator is of type ulong, a compile-time error occurs. An exception is the rule that permits the long value -9223372036854775808 ( $-2^{63}$ ) to be written as a decimal integer literal (§7.4.5.3)

• Floating-point negation:

```
float operator -(float x);
double operator -(double x);
```

The result is the value of x with its sign inverted. If x is NaN, the result is also NaN.

• Decimal negation:

```
decimal operator -(decimal x);
```

The result is computed by subtracting x from zero. Decimal negation is equivalent to using the unary minus operator of type System.Decimal.

Lifted (§12.4.8) forms of the unlifted predefined unary minus operators defined above are also predefined.

## 12.8.4 Logical negation operator

For an operation of the form !x, unary operator overload resolution (§12.4.4) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

bool operator !(bool x);

This operator computes the logical negation of the operand: If the operand is true, the result is false. If the operand is false, the result is true.

Lifted (§12.4.8) forms of the unlifted predefined logical negation operator defined above are also predefined.

#### 12.8.5 Bitwise complement operator

For an operation of the form  $\sim x$ , unary operator overload resolution (§12.4.4) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

For each of these operators, the result of the operation is the bitwise complement of x.

Every enumeration type E implicitly provides the following bitwise complement operator:

E operator  $\sim$ (E x);

The result of evaluating  $\sim x$ , where x is an expression of an enumeration type E with an underlying type U, is exactly the same as evaluating (E) ( $\sim$ (U)x), except that the conversion to E is always performed as if in an unchecked context (§12.7.14).

Lifted (§12.4.8) forms of the unlifted predefined bitwise complement operators defined above are also predefined.

## 12.8.6 Prefix increment and decrement operators

pre-increment-expression: ++ unary-expression pre-decrement-expression:

-- unary-expression

The operand of a prefix increment or decrement operation shall be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer shall have both a get and a set accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (§12.4.4) is applied to select a specific operator implementation. Predefined ++ and -- operators exist for the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any enum type. The predefined ++ operators return the value produced by adding 1 to the operand, and the predefined -- operators return the value produced by subtracting 1 from the operand. In a checked context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a System.OverflowException is thrown.

There shall be an implicit conversion from the return type of the selected unary operator to the type of the *primary-expression*, otherwise a compile-time error occurs.

The run-time processing of a prefix increment or decrement operation of the form ++x or --x consists of the following steps:

- If x is classified as a variable:
  - x is evaluated to produce the variable.
  - The value of x is converted to the operand type of the selected operator and the operator is invoked with this value as its argument.
  - The value returned by the operator is converted to the type of x. The resulting value is stored in the location given by the evaluation of x.
  - and becomes the result of the operation.
- If x is classified as a property or indexer access:
  - The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent get and set accessor invocations.
  - The get accessor of x is invoked.
  - The value returned by the get accessor is converted to the operand type of the selected operator and operator is invoked with this value as its argument.
  - The value returned by the operator is converted to the type of x. The set accessor of x is invoked with this value as its value argument.

• This value also becomes the result of the operation.

The ++ and -- operators also support postfix notation (§12.7.10). Typically, the result of x++ or x-- is the value of x *before* the operation, whereas the result of ++x or --x is the value of x *after* the operation. In either case, x itself has the same value after the operation.

An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

Lifted (§12.4.8) forms of the unlifted predefined prefix increment and decrement operators defined above are also predefined.

# 12.8.7 Cast expressions

A cast-expression is used to convert explicitly an expression to a given type.

cast-expression: ( type ) unary-expression

A *cast-expression* of the form (T) E, where T is a *type* and E is a *unary-expression*, performs an explicit conversion (§11.3) of the value of E to type T. If no explicit conversion exists from E to T, a binding-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if E denotes a variable.

The grammar for a *cast-expression* leads to certain syntactic ambiguities. [*Example*: The expression (x)–y could either be interpreted as a *cast-expression* (a cast of –y to type x) or as an *additive-expression* combined with a *parenthesized-expression* (which computes the value x - y). *end example*]

To resolve *cast-expression* ambiguities, the following rule exists: A sequence of one or more *tokens* (§7.4) enclosed in parentheses is considered the start of a *cast-expression* only if at least one of the following are true:

- The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
- The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing parentheses is the token "~", the token "!", the token "(", an *identifier* (§7.4.3), a *literal* (§7.4.5), or any *keyword* (§7.4.4) except as and is.

The term "correct grammar" above means only that the sequence of tokens shall conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. [*Example*: If x and y are identifiers, then x.y is correct grammar for a type, even if x.y doesn't actually denote a type. *end example*]

[*Note*: From the disambiguation rule, it follows that, if x and y are identifiers, (x)y, (x)(y), and (x)(-y) are *cast-expressions*, but (x)-y is not, even if x identifies a type. However, if x is a keyword that identifies a predefined type (such as int), then all four forms are *cast-expressions* (because such a keyword could not possibly be an expression by itself). *end note*]

## 12.8.8 Await expressions

## 12.8.8.1 General

The await operator is used to suspend evaluation of the enclosing async function until the asynchronous operation represented by the operand has completed.

await-expression: await unary-expression

An *await-expression* is only allowed in the body of an async function (§15.15). Within the nearest enclosing async function, an *await-expression* shall not occur in these places:

• Inside a nested (non-async) anonymous function

- In a catch or finally block of a *try-statement*
- Inside the block of a *lock-statement*
- In an anonymous function conversion to an expression tree type (§11.7.3)
- In an unsafe context

[*Note*: An *await-expression* cannot occur in most places within a *query-expression*, because those are syntactically transformed to use non-async lambda expressions. *end note*]

Inside an async function, await shall not be used as an *available-identifier* although the verbatim identifier @await may be used. There is therefore no syntactic ambiguity between await-expressions and various expressions involving identifiers. Outside of async functions, await acts as a normal identifier.

The operand of an *await-expression* is called the **task**. It represents an asynchronous operation that may or may not be complete at the time the *await-expression* is evaluated. The purpose of the await operator is to suspend execution of the enclosing async function until the awaited task is complete, and then obtain its outcome.

## 12.8.8.2 Awaitable expressions

The task of an await expression is required to be *awaitable*. An expression *t* is awaitable if one of the following holds:

- *t* is of compile-time type dynamic
- *t* has an accessible instance or extension method called GetAwaiter with no parameters and no type parameters, and a return type A for which all of the following hold:
  - A implements the interface System.Runtime.CompilerServices.INotifyCompletion (hereafter known as INotifyCompletion for brevity)
  - A has an accessible, readable instance property IsCompleted of type bool
  - A has an accessible instance method GetResult with no parameters and no type parameters

The purpose of the GetAwaiter method is to obtain an *awaiter* for the task. The type A is called the *awaiter type* for the await expression.

The purpose of the IsCompleted property is to determine if the task is already complete. If so, there is no need to suspend evaluation.

The purpose of the INotifyCompletion.OnCompleted method is to sign up a "continuation" to the task; i.e., a delegate (of type System.Action) that will be invoked once the task is complete.

The purpose of the GetResult method is to obtain the outcome of the task once it is complete. This outcome may be successful completion, possibly with a result value, or it may be an exception which is thrown by the GetResult method.

#### 12.8.8.3 Classification of await expressions

The expression await t is classified the same way as the expression

(t).GetAwaiter().GetResult().Thus, if the return type of GetResult is void, the *await-expression* is classified as nothing. If it has a non-void return type  $\tau$ , the *await-expression* is classified as a value of type  $\tau$ .

#### 12.8.8.4 Run-time evaluation of await expressions

At run-time, the expression await t is evaluated as follows:

- An awaiter *a* is obtained by evaluating the expression (*t*).GetAwaiter().
- A bool *b* is obtained by evaluating the expression (*a*).IsCompleted.

- If b is false then evaluation depends on whether a implements the interface System.Runtime.CompilerServices.ICriticalNotifyCompletion (hereafter known as ICriticalNotifyCompletion for brevity). This check is done at binding time; i.e., at run-time if a has the compile-time type dynamic, and at compile-time otherwise. Let r denote the resumption delegate (§15.15):
  - If a does not implement ICriticalNotifyCompletion, then the expression
     ((a) as INotifyCompletion).OnCompleted(r) is evaluated.
  - If a does implement ICriticalNotifyCompletion, then the expression
     ((a) as ICriticalNotifyCompletion).UnsafeOnCompleted(r) is evaluated.
  - Evaluation is then suspended, and control is returned to the current caller of the async function.
- Either immediately after (if b was true), or upon later invocation of the resumption delegate (if b was false), the expression (a).GetResult() is evaluated. If it returns a value, that value is the result of the await-expression. Otherwise, the result is nothing.

An awaiter's implementation of the interface methods INotifyCompletion.OnCompleted and ICriticalNotifyCompletion.UnsafeOnCompleted should cause the delegate r to be invoked at most once. Otherwise, the behavior of the enclosing async function is undefined.

# 12.9 Arithmetic operators

## 12.9.1 General

The \*, /, %, +, and – operators are called the arithmetic operators.

multiplicative-expression: unary-expression multiplicative-expression \* unary-expression multiplicative-expression / unary-expression multiplicative-expression % unary-expression additive-expression: multiplicative-expression additive-expression + multiplicative-expression additive-expression - multiplicative-expression

If an operand of an arithmetic operator has the compile-time type dynamic, then the expression is dynamically bound (§12.3.3). In this case, the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

## 12.9.2 Multiplication operator

For an operation of the form x \* y, binary operator overload resolution (§12.4.5) 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 IEC 60559 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, rounded to the nearest representable value. If the magnitude of the result is too large for the destination type, z is infinity. Because of rounding, z may be zero even though neither x nor y is zero.

	+y	-у	+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							

(Except were otherwise noted, in the floating-point tables in 12.9.2-12.9.6 the use of "+" means the value is positive; the use of "-" means the value is negative; and the lack of a sign means the value may be positive or negative or has no sign (NaN).)

• Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the magnitude of the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. Because of rounding, the result may be zero even though neither operand 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.

Lifted (§12.4.8) forms of the unlifted predefined multiplication operators defined above are also predefined.

## 12.9.3 Division operator

For an operation of the form x / y, binary operator overload resolution (§12.4.5) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of x and y.

• Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

If the value of the right operand is zero, a System.DivideByZeroException is thrown.

The division rounds the result towards zero. Thus the absolute value of the result is the largest possible integer that is less than or equal to the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have the same sign and zero or negative when the two operands have opposite signs.

If the left operand is the smallest representable int or long value and the right operand is -1, an overflow occurs. In a checked context, this causes a System.ArithmeticException (or a subclass thereof) to be thrown. In an unchecked context, it is implementation-defined as to whether a System.ArithmeticException (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand.

• Floating-point division:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

The quotient is computed according to the rules of IEC 60559 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, rounded to the nearest representable value.

	+y	-у	+0	-0	+∞	-∞	NaN
+x	+Z	-z	+∞	-∞	+0	-0	NaN
-x	-z	+Z	-∞	+∞	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+∞	+∞	-∞	+∞	-∞	NaN	NaN	NaN
-∞	-∞	+∞	-∞	+∞	NaN	NaN	NaN
NaN							

• Decimal division:

decimal operator /(decimal x, decimal y);

If the value of the right operand is zero, a System.DivideByZeroException is thrown. If the magnitude of the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. Because of rounding, the result may be zero even though the first operand is not zero. The scale of the result, before any rounding, is the closest scale to the preferred scale that will preserve a result equal to the exact result. The preferred scale is the scale of x less the scale of y.

Decimal division is equivalent to using the division operator of type System.Decimal.

Lifted (§12.4.8) forms of the unlifted predefined division operators defined above are also predefined.

#### 12.9.4 Remainder operator

For an operation of the form x % y, binary operator overload resolution (§12.4.5) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between x and y.

• Integer remainder:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

The result of x % y is the value produced by x - (x / y) \* y. If y is zero, a System.DivideByZeroException is thrown.

If the left operand is the smallest int or long value and the right operand is -1, a System.OverflowException is thrown if and only if x / y would throw an exception.

• Floating-point remainder:

float operator %(float x, float y);
double operator %(double x, double y);

The following table lists the results of all possible combinations of 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 and is computed as x - n + y, where n is the largest possible integer that is less than or equal to x / y. This method of computing the remainder is analogous to that used for integer operands, but differs from the IEC 60559 definition (in which n is the integer closest to x / y).

	+у	-у	+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							

• Decimal remainder:

decimal operator %(decimal x, decimal y);

If the value of the right operand is zero, a System.DivideByZeroException is thrown. It is implementation-defined when a System.ArithmeticException (or a subclass thereof) is thrown. A conforming implementation shall not throw an exception for x % y in any case where x / y does not throw an exception. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of x.

Decimal remainder is equivalent to using the remainder operator of type System.Decimal.

[*Note*: These rules ensure that for all types, the result never has the opposite sign of the left operand. *end note*]

Lifted (§12.4.8) forms of the unlifted predefined remainder operators defined above are also predefined.

#### 12.9.5 Addition operator

For an operation of the form x + y, binary operator overload resolution (§12.4.5) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or both operands are of type string, the predefined addition operators concatenate the string representation of the operands.

• Integer addition:

int operator +(int x, int y); uint operator +(uint x, uint y); long operator +(long x, long y); ulong operator +(ulong x, ulong y In a checked context, if the sum is outside the range of the result type, a

System.OverflowException is thrown. In an unchecked context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

• Floating-point addition:

float operator +(float x, float y);
double operator +(double x, double y);

The sum is computed according to the rules of IEC 60559 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 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.

	У	+0	-0	+∞	-∞	NaN
х	z	х	х	+∞	-∞	NaN
+0	У	+0	+0	+∞	-∞	NaN
-0	У	+0	-0	+∞	-∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
-∞	-∞	-∞	-∞	NaN	-∞	NaN
NaN						

• Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

If the magnitude of 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);
```

At run-time these 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);

These overloads of the binary + operator perform string concatenation. If an operand of string concatenation is null, an empty string is substituted. Otherwise, any non-string operand 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
    }
}</pre>
```

The output shown in the comments is the typical result on a US-English system. The precise output might depend on the regional settings of the execution environment. The string-concatenation operator itself behaves the same way in each case, but the ToString methods implicitly called during execution might be affected by regional settings. *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.

• Delegate combination. Every delegate type implicitly provides the following predefined operator, where D is the delegate type:

D operator +(D x, D y);

If the first operand is null, the result of the operation is the value of the second operand (even if that is also null). Otherwise, if the second operand is null, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance whose invocation list consists of the elements in the invocation list of the first operand, followed by the elements in the invocation list of the resulting delegate is the concatenation of the invocation lists of the two operands. [*Note*: For examples of delegate combination, see §12.9.6 and §20.6. Since System.Delegate is not a delegate type, operator + is not defined for it. *end note*]

Lifted (§12.4.8) forms of the unlifted predefined addition operators defined above are also predefined.

## 12.9.6 Subtraction operator

For an operation of the form x - y, binary operator overload resolution (§12.4.5) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined subtraction operators are listed below. The operators all subtract y from x.

• Integer subtraction:

int operator -(int x, int y); uint operator -(uint x, uint y); long operator -(long x, long y); ulong operator -(ulong x, ulong y

In a checked context, if the difference is outside the range of the result type, a System.OverflowException is thrown. In an unchecked context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

• Floating-point subtraction:

```
float operator -(float x, float y);
double operator -(double x, double y);
```

The difference is computed according to the rules of IEC 60559 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaNs. In the table, x 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 x - y is too large to represent in the destination type, z is an infinity with the same sign as x - y.

	У	+0	-0	+∞	-∞	NaN
x	z	х	х	-∞	+∞	NaN
+0	-у	+0	+0	-∞	+∞	NaN
-0	-у	-0	+0	-∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
-∞	-∞	-∞	-∞	-∞	NaN	NaN
NaN						

(In the above table the -y entries denote the *negation* of y, not that the value is negative.)

• Decimal subtraction:

decimal operator -(decimal x, decimal y);

If the magnitude of the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type System.Decimal.

• Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where E is the enum type, and U is the underlying type of E:

U operator -(E x, E y);

This operator is evaluated exactly as (U)((U)x - (U)y). In other words, the operator computes the difference between the ordinal values of x and y, and the type of the result is the underlying type of the enumeration.

E operator -(E x, U y);

This operator is evaluated exactly as (E)((U)x - y). In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

• Delegate removal. Every delegate type implicitly provides the following predefined operator, where D is the delegate type:

```
D operator -(D x, D y);
```

If the first operand is null, the result of the operation is null. Otherwise, if the second operand is null, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists (§20.2) having one or more entries, and the result is a new invocation list consisting of the first operand's list with the second operand's entries removed from it, provided the second operand's list is a proper contiguous sublist of the first's. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator (§12.11.9).) Otherwise, the result is the value of the left operand. Neither of the operands' lists is changed in the process. If the second operand's list matches multiple sublists of contiguous entries in the first operand's list, the right-most matching sublist of contiguous entries is removed. If removal results in an empty list, the result is null. [*Example*:

```
delegate void D(int x);
class C
{
   public static void M1(int i) { /* ... */ }
public static void M2(int i) { /* ... */ }
}
class Test
   static void Main() {
    D cd1 = new D(C.M1);
    D cd2 = new D(C.M2);
                                                // M1 + M2 + M2 + M1
       D cd3 = cd1 + cd2 + cd2 + cd1;
       cd3 -= cd1;
                                                // => M1 + M2 + M2
       cd3 = cd1 + cd2 + cd2 + cd1;
                                                // M1 + M2 + M2 + M1
       cd3 -= cd1 + cd2;
                                                // => M2 + M1
                                                // M1 + M2 + M2 + M1
// => M1 + M1
       cd3 = cd1 + cd2 + cd2 + cd1;
       cd3 -= cd2 + cd2;
       cd3 = cd1 + cd2 + cd2 + cd1;
                                                // M1 + M2 + M2 + M1
       cd3 -= cd2 + cd1;
                                                // => M1 + M2
       cd3 = cd1 + cd2 + cd2 + cd1;
                                                // M1 + M2 + M2 + M1
       cd3 -= cd1 + cd1;
                                                // => M1 + M2 + M2 + M1
   }
}
```

end example]

Lifted (§12.4.8) forms of the unlifted predefined subtraction operators defined above are also predefined.

## 12.10 Shift operators

The << and >> operators are used to perform bit-shifting operations.

shift-expression: additive-expression shift-expression << additive-expression shift-expression right-shift additive-expression

If an operand of a *shift-expression* has the compile-time type dynamic, then the expression is dynamically bound (§12.3.3). In this case, the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

For an operation of the form  $x \ll \text{count or } x \gg \text{count}$ , binary operator overload resolution (§12.4.5) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand shall always be the class or struct containing the operator declaration, and the type of the second operand shall always be int.

The predefined shift operators are listed below.

• Shift left:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);</pre>
```

The << operator shifts x left by a number of bits computed as described below.

The high-order bits outside the range of the result type of x are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

• Shift right:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

The >> operator shifts x right by a number of bits computed as described below.

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.

When x is of type uint or ulong, the low-order bits of x are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

For the predefined operators, the number of bits to shift is computed as follows:

- When the type of x is int or uint, the shift count is given by the low-order five bits of count. In other words, the shift count is computed from count & 0x1F.
- When the type of x is long or ulong, the shift count is given by the low-order six bits of count. In other words, the shift count is computed from count & 0x3F.

If the resulting shift count is zero, the shift operators simply return the value of x.

Shift operations never cause overflows and produce the same results in checked and unchecked contexts.

When the left operand of the >> operator is of a signed integral type, the operator performs an *arithmetic* shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the >> operator is of an unsigned integral type, the operator performs a *logical* shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. [*Example*: If x is a variable of type int, the operation unchecked((int)((uint)x >> y)) performs a logical shift right of x. *end example*]

Lifted (§12.4.8) forms of the unlifted predefined shift operators defined above are also predefined.

## 12.11 Relational and type-testing operators

#### 12.11.1 General

The ==, !=, <, >, <=, >=, is, and as operators are called the relational and type-testing operators.

```
relational-expression:

shift-expression

relational-expression < shift-expression

relational-expression > shift-expression

relational-expression <= shift-expression

relational-expression >= shift-expression

relational-expression is type

relational-expression as type

equality-expression:

relational-expression

equality-expression == relational-expression

equality-expression != relational-expression
```

The is operator is described in §12.11.11 and the as operator is described in §12.11.12.

The ==, !=, <, >, <= and >= operators are *comparison operators*.

If an operand of a comparison operator has the compile-time type dynamic, then the expression is dynamically bound (§12.3.3). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

For an operation of the form x *op* y, where *op* is a comparison operator, overload resolution (\$12.4.5) 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. If both operands of an *equality-expression* are the null literal, then overload resolution is not performed and the expression evaluates to a constant value of true or false according to whether the operator is == or !=.

The predefined comparison operators are described in the following subclauses. All predefined comparison operators return a result of type bool, as described in the following table.

Operation	Result			
x == y	true if x is equal to y, false otherwise			
x != y	true if x is not equal to y, false otherwise			
x < y	true if x is less than y, false otherwise			
x > y	true if x is greater than y, false otherwise			
x <= y	true if x is less than or equal to y, false otherwise			
x >= y	true if x is greater than or equal to y, false otherwise			

#### 12.11.2 Integer comparison operators

The predefined integer comparison operators are:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong \dot{x}, ulong \dot{y});
bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y)
bool operator !=(ulong x, ulong y);
bool operator <(int x, int y);
bool operator <(uint x, uint y);</pre>
bool operator <(long x, long y)
bool operator <(ulong \dot{x}, ulong \dot{y});
bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);
bool operator \leq (int x, int y);
bool operator <=(uint x, uint y);</pre>
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);</pre>
bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Each of these operators compares the numeric values of the two integer operands and returns a bool value that indicates whether the particular relation is true or false.

Lifted (§12.4.8) forms of the unlifted predefined integer comparison operators defined above are also predefined.

#### 12.11.3 Floating-point comparison operators

The predefined floating-point comparison operators are:

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);
bool operator !=(float x, float y);
bool operator !=(double x, double y);
bool operator <(float x, float y);
bool operator <(double x, double y);
bool operator >(float x, float y);
bool operator >(double x, double y);
bool operator <=(float x, float y);
bool operator <=(float x, float y);
bool operator >=(float x, float y);
```

The operators compare the operands according to the rules of the IEC 60559 standard:

If either operand is NaN, the result is false for all operators except !=, for which the result is true. For any two operands, x != y always produces the same result as !(x == y). However, when one or both operands are NaN, the <, >, <=, and >= operators do *not* produce the same results as the logical negation of the opposite operator. [*Example*: If either of x and y is NaN, then x < y is false, but !(x == y) is true. *end example*]

• When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

 $-\infty < -\max < ... < -\min < -0.0 == +0.0 < +\min < ... < +\max < +\infty$ 

where min and max are the smallest and largest positive finite values that can be represented in the given floating-point format. Notable effects of this ordering are:

- Negative and positive zeros are considered equal.
- A negative infinity is considered less than all other values, but equal to another negative infinity.
- o A positive infinity is considered greater than all other values, but equal to another positive infinity.

Lifted (§12.4.8) forms of the unlifted predefined floating-point comparison operators defined above are also predefined.

#### 12.11.4 Decimal comparison operators

The predefined decimal comparison operators are:

bool	operator	==(decimal x, decimal y);
		<pre>!=(decimal x, decimal y);</pre>
		<(decimal x, decimal y);
		<pre>&gt;(decimal_x, decimal_y);</pre>
		<=(decimal x, decimal y);
bool	operator	<pre>&gt;=(decimal x, decimal y);</pre>

Each of these operators compares the numeric values of the two decimal operands and returns a bool value that indicates whether the particular relation is true or false. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type System.Decimal.

Lifted (§12.4.8) forms of the unlifted predefined decimal comparison operators defined above are also predefined.

#### 12.11.5 Boolean equality operators

The predefined Boolean equality operators are:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

The result of == is true if both x and y are true or if both x and y are false. Otherwise, the result is false.

The result of != is false if both x and y are true or if both x and y are false. Otherwise, the result is true. When the operands are of type bool, the != operator produces the same result as the  $\land$  operator.

Lifted (§12.4.8) forms of the unlifted predefined Boolean equality operators defined above are also predefined.

#### 12.11.6 Enumeration comparison operators

Every enumeration type implicitly provides the following predefined comparison operators

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating x op y, where x and y are expressions of an enumeration type E with an underlying type U, and op is one of the comparison operators, is exactly the same as evaluating ((U)x) op ((U)y). In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

Lifted (§12.4.8) forms of the unlifted predefined enumeration comparison operators defined above are also predefined.

### 12.11.7 Reference type equality operators

Every class type C implicitly provides the following predefined reference type equality operators:

```
bool operator ==(C x, C y);
bool operator !=(C x, C y);
```

unless predefined equality operators otherwise exist for C (for example, when C is string or System.Delegate).

The operators return the result of comparing the two references for equality or non-equality. operator == returns true if and only if x and y refer to the same instance or are both null, while operator != returns true if and only if operator == with the same operands would return false.

In addition to normal applicability rules (§12.6.4.2), the predefined reference type equality operators require one of the following in order to be applicable:

- Both operands are a value of a type known to be a *reference-type* or the literal null. Furthermore, an explicit identity or reference conversion (§11.3.5) exists from either operand to the type of the other operand.
- One operand is the literal null, and the other operand is a value of type T where T is a *type*parameter that is not known to be a value type, and does not have the value type constraint.
  - If at runtime T is a non-nullable value type, the result of == is false and the result of != is true.
  - If at runtime T is a nullable value type, the result is computed from the HasValue property of the operand, as described in (§12.11.10).
  - o If at runtime T is a reference type, the result is true if the operand is null, and false otherwise.

Unless one of these conditions is true, a binding-time error occurs.

[Note: Notable implications of these rules are:

- It is a binding-time error to use the predefined reference type equality operators to compare two references that are known to be different at binding-time. For example, if the binding-time types of the operands are two class types, and if neither derives from the other, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a binding-time error.
- The predefined reference type equality operators do not permit value type operands to be compared (except when type parameters are compared to null, which is handled specially).
- Operands of predefined reference type equality operators are never boxed. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.

For an operation of the form x = y or x != y, if any applicable user-defined operator = or operator != exists, the operator overload resolution rules (§12.4.5) will select that operator instead of the predefined reference type equality operator. It is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type object. *end note*]

[*Example*: The following example checks whether an argument of an unconstrained type parameter type is null.

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

The x == null construct is permitted even though T could represent a non-nullable value type, and the result is simply defined to be false when T is a non-nullable value type. *end example*]

For an operation of the form x == y or x != y, if any applicable operator == or operator != exists, the operator overload resolution (§12.4.5) rules will select that operator instead of the predefined reference type equality operator. [*Note*: It is always possible to select the predefined reference type equality operator by explicitly casting both of the operands to type object. *end note*]

[Example: The example

```
using System;
class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine(object)s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine(s == (object)t);
    }
}
```

produces the output

True False False False

The s and t variables refer to two distinct string instances containing the same characters. The first comparison outputs True because the predefined string equality operator (§12.11.8) is selected when both operands are of type string. The remaining comparisons all output False because the overload of operator== in the string type is not applicable when either operand has a binding-time type of object.

Note that the above technique is not meaningful for value types. The example

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

outputs False because the casts create references to two separate instances of boxed int values. *end example*]

# 12.11.8 String equality operators

The predefined string equality operators are:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two string values are considered equal when one of the following is true:

- Both values are null.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string values rather than string references. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. [*Note*: As described in §12.11.7, the reference type equality operators can be used to compare string references instead of string values. *end note*]

#### 12.11.9 Delegate equality operators

The predefined delegate equality operators are:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Two delegate instances are considered equal as follows:

- If either of the delegate instances is null, they are equal if and only if both are null.
- If the delegates have different run-time type, they are never equal.
- If both of the delegate instances have an invocation list (§20.2), those instances are equal if and only if their invocation lists are the same length, and each entry in one's invocation list is equal (as defined below) to the corresponding entry, in order, in the other's invocation list.

The following rules govern the equality of invocation list entries:

- If two invocation list entries both refer to the same static method then the entries are equal.
- If two invocation list entries both refer to the same non-static method on the same target object (as defined by the reference equality operators) then the entries are equal.
- Invocation list entries produced from evaluation of semantically identical anonymous functions (§12.16) with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.

If operator overload resolution resolves to either delegate equality operator, and the binding-time types of both operands are delegate types as described in §20 rather than System.Delegate, and there is no identity conversion between the binding-type operand types, a binding-time error occurs.

[*Note*: This rule prevents comparisons which can never consider non-null values as equal due to being references to instances of different types of delegates. *end note*]

# 12.11.10 Equality operators between nullable value types and the null literal

The == and != operators permit one operand to be a value of a nullable value type and the other to be the null literal, even if no predefined or user-defined operator (in unlifted or lifted form) exists for the operation.

For an operation of one of the forms

x == null null == x x != null null != x

where x is an expression of a nullable value type, if operator overload resolution (§12.4.5) fails to find an applicable operator, the result is instead computed from the HasValue property of x. Specifically, the first two forms are translated into <code>!x.HasValue</code>, and the last two forms are translated into <code>x.HasValue</code>.

# 12.11.11 The is operator

The is operator is used to check if the run-time type of an object is compatible with a given type. The check is performed at runtime. The result of the operation E is T, where E is an expression and T is a type other than dynamic, is a Boolean value indicating whether E is non-null and can successfully be converted to type T by a reference conversion, a boxing conversion, an unboxing conversion, a wrapping conversion, or an unwrapping conversion.

The operation is evaluated as follows:

- 1. If E is an anonymous function, a compile-time error occurs
- 2. If E is a method group or the null literal, of if the value of E is null, the result is false.
- 3. Otherwise:
- 4. Let R be the runtime type of E.
- 5. Let D be derived from R as follows:
- 6. If R is a nullable value type, D is the underlying type of R.
- 7. Otherwise, D is R.
- 8. The result depends on D and T as follows:
- 9. If T is a reference type, the result is true if:
  - D and T are the same type,
  - $\circ~$  D is a reference type and an implicit reference conversion from D to T exists, or
  - Either: D is a value type and a boxing conversion from D to T exists. Or: D is a value type and T is an interface type implemented by D.
- 10. If T is a nullable value type, the result is true if D is the underlying type of T.
- 11. If T is a non-nullable value type, the result is true if D and T are the same type.
- 12. Otherwise, the result is false.

User defined conversions are not considered by the is operator.

[*Note*: As the is operator is evaluated at runtime, all type arguments have been substituted and there are no open types (§9.4.3) to consider. *end note*]

[*Note*: The is operator can be understood in terms of compile-time types and conversions as follows, where C is the compile-time type of E:

- If the compile-time type of e is the same as T, or if an implicit reference conversion (§11.2.7), boxing conversion (§11.2.8), wrapping conversion (§11.6), or an explicit unwrapping conversion (§11.6) exists from the compile-time type of E to T:
  - If C is of a non-nullable value type, the result of the operation is true.

- $\circ$  Otherwise, the result of the operation is equivalent to evaluating E != null.
- Otherwise, if an explicit reference conversion (§11.3.5) or unboxing conversion (§11.3.6) exists from C to T, or if C or T is an open type (§9.4.3), then runtime checks as above must be peformed.
- Otherwise, no reference, boxing, wrapping, or unwrapping conversion of E to type T is possible, and the result of the operation is false.

A compiler may implement optimisations based on the compile-time type. end note]

#### 12.11.12 The as operator

The as operator is used to explicitly convert a value to a given reference type or nullable value type. Unlike a cast expression (§12.8.7), the as operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is null.

In an operation of the form E as T, E shall be an expression and T shall be a reference type, a type parameter known to be a reference type, or a nullable value type. Furthermore, at least one of the following shall be true, or otherwise a compile-time error occurs:

- An identity (§11.2.2), implicit nullable (§11.2.5), implicit reference (§11.2.7), boxing (§11.2.8), explicit nullable (§11.3.4), explicit reference (§11.3.5), or wrapping (§9.3.11) conversion exists from E to T.
- The type of E or T is an open type.
- E is the null literal.

If the compile-time type of E is not dynamic, the operation E as T produces the same result as

E is T ? (T)(E) : (T)null

except that E is only evaluated once. The compiler can be expected to optimize E as T to perform at most one runtime type check as opposed to the two runtime type checks implied by the expansion above.

If the compile-time type of E is dynamic, unlike the cast operator the as operator is not dynamically bound (§12.3.3). Therefore the expansion in this case is:

E is T ? (T)(object)(E) : (T)null

Note that some conversions, such as user defined conversions, are not possible with the as operator and should instead be performed using cast expressions.

[Example: In the example

```
class X
{
    public string F(object o) {
        return o as string; // OK, string is a reference type
    }
    public T G<T>(object o) where T: Attribute {
        return o as T; // Ok, T has a class constraint
    }
    public U H<U>(object o) {
        return o as U; // Error, U is unconstrained
    }
}
```

the type parameter T of G is known to be a reference type, because it has the class constraint. The type parameter U of H is not however; hence the use of the as operator in H is disallowed. *end example*]

### 12.12 Logical operators

## 12.12.1 General

The &, ^, and | operators are called the logical operators.

and-expression: equality-expression and-expression & equality-expression exclusive-or-expression: and-expression exclusive-or-expression ^ and-expression inclusive-or-expression: exclusive-or-expression

inclusive-or-expression | exclusive-or-expression

If an operand of a logical operator has the compile-time type dynamic, then the expression is dynamically bound (§12.3.3). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

For an operation of the form x *op* y, where *op* is one of the logical operators, overload resolution (§12.4.5) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following subclauses.

## 12.12.2 Integer logical operators

The predefined integer logical operators are:

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);
int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator ^(int x, int y);
uint operator ^(int x, uint y);
long operator ^(long x, long y);
uint operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

The & operator computes the bitwise logical AND of the two operands, the | operator computes the bitwise logical OR of the two operands, and the ^ operator computes the bitwise logical exclusive OR of the two operands. No overflows are possible from these operations.

Lifted (§12.4.8) forms of the unlifted predefined integer logical operators defined above are also predefined.

#### 12.12.3 Enumeration logical operators

Every enumeration type E implicitly provides the following predefined logical operators:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

The result of evaluating x *op* y, where x and y are expressions of an enumeration type E with an underlying type U, and *op* is one of the logical operators, is exactly the same as evaluating  $(E)((U) \times op (U) y)$ . In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

Lifted (§12.4.8) forms of the unlifted predefined enumeration logical operators defined above are also predefined.

## 12.12.4 Boolean logical operators

The predefined Boolean logical operators are:

boo1	operator	&(bool	х,	boo1	y);
boo1	operator	(boo]	х,	boo1	y);
bool	operator	^(bool	х,	bool	y);

The result of x & y is true if both x and y are true. Otherwise, the result is false.

The result of  $x \mid y$  is true if either x or y is true. Otherwise, the result is false.

The result of  $x \land y$  is true if x is true and y is false, or x is false and y is true. Otherwise, the result is false. When the operands are of type bool, the  $\land$  operator computes the same result as the != operator.

#### 12.12.5 Nullable Boolean & and | operators

The nullable Boolean type bool? can represent three values, true, false, and null.

As with the other binary operators, lifted forms of the logical operators & and | (§12.12.4) are also predefined:

bool? operator &(bool? x, bool? y); bool? operator |(bool? x, bool? y);

The semantics of the lifted & and | operators are defined by the following table:

x	У	х&у	x   y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

[*Note*: The bool? type is conceptually similar to the three-valued type used for Boolean expressions in SQL. The table above follows the same semantics as SQL, whereas applying the rules of §12.4.8 to the & and | operators would not. The rules of §12.4.8 already provide SQL-like semantics for the lifted ^ operator. *end note*]

## 12.13 Conditional logical operators

## 12.13.1 General

The && and || operators are called the conditional logical operators. They are also called the "short-circuiting" logical operators.

```
conditional-and-expression:

inclusive-or-expression

conditional-and-expression && inclusive-or-expression

conditional-or-expression:

conditional-and-expression

conditional-or-expression || conditional-and-expression
```

The && and || operators are conditional versions of the & and | operators:

- The operation x && y corresponds to the operation x & y, except that y is evaluated only if x is not false.
- The operation x || y corresponds to the operation x | y, except that y is evaluated only if x is not true.

[*Note*: The reason that short circuiting uses the 'not true' and 'not false' conditions is to enable userdefined conditional operators to define when short circuiting applies. User-defined types could be in a state where operator true returns false and operator false returns false. In those cases, neither && nor || would short circuit. *end note*]

If an operand of a conditional logical operator has the compile-time type dynamic, then the expression is dynamically bound (§12.3.3). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

An operation of the form x & y or x || y is processed by applying overload resolution (§12.4.5) as if the operation was written x & y or x | y. Then,

- If overload resolution fails to find a single best operator, or if overload resolution selects one of the predefined integer logical operators or nullable Boolean logical operators (§12.12.5), a binding-time error occurs.
- Otherwise, if the selected operator is one of the predefined Boolean logical operators (§12.12.4), the operation is processed as described in §12.13.2.
- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in §12.13.3.

It is not possible to directly overload the conditional logical operators. However, because the conditional logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical operators are, with certain restrictions, also considered overloads of the conditional logical operators. This is described further in §12.13.3.

# 12.13.2 Boolean conditional logical operators

When the operands of && or || are of type bool, or when the operands are of types that do not define an applicable operator & or operator |, but do define implicit conversions to bool, the operation is processed as follows:

- The operation x && y is evaluated as x ? y : false. In other words, x is first evaluated and converted to type bool. Then, if x is true, y is evaluated and converted to type bool, and this becomes the result of the operation. Otherwise, the result of the operation is false.
- The operation x || y is evaluated as x ? true : y. In other words, x is first evaluated and converted to type bool. Then, if x is true, the result of the operation is true. Otherwise, y is evaluated and converted to type bool, and this becomes the result of the operation.

## 12.13.3 User-defined conditional logical operators

When the operands of && or || are of types that declare an applicable user-defined operator & or operator |, both of the following shall be true, where T is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator shall be T. In other words, the operator shall compute the logical AND or the logical OR of two operands of type T, and shall return a result of type T.
- T shall contain declarations of operator true and operator false.

A binding-time error occurs if either of these requirements is not satisfied. Otherwise, the && or || operation is evaluated by combining the user-defined operator true or operator false with the selected user-defined operator:

- The operation x && y is evaluated as T.false(x) ? x : T.&(x, y), where T.false(x) is an invocation of the operator false declared in T, and T.&(x, y) is an invocation of the selected operator &. In other words, x is first evaluated and operator false is invoked on the result to determine if x is definitely false. Then, if x is definitely false, the result of the operator & is invoked on the value previously computed for x. Otherwise, y is evaluated, and the selected operator & is invoked on the value previously computed for x and the value computed for y to produce the result of the operation.
- The operation x || y is evaluated as T.true(x) ? x : T. | (x, y), where T.true(x) is an invocation of the operator true declared in T, and T. | (x, y) is an invocation of the selected operator |. In other words, x is first evaluated and operator true is invoked on the result to determine if x is definitely true. Then, if x is definitely true, the result of the operator | is invoked on the value previously computed for x. Otherwise, y is evaluated, and the selected operator | is invoked on the value previously computed for x and the value computed for y to produce the result of the operation.

In either of these operations, the expression given by x is only evaluated once, and the expression given by y is either not evaluated or evaluated exactly once.

# 12.14 The null coalescing operator

The ?? operator is called the null coalescing operator.

null-coalescing-expression: conditional-or-expression conditional-or-expression ?? null-coalescing-expression

A null coalescing expression of the form a ?? b requires a to be the null literal (57.4.5.7), or to be of a nullable value type or reference type. If a is non-null, the result of a ?? b is a; otherwise, the result is b. The operation evaluates b only if a is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. [*Example*: An expression of the form a ?? b ?? c is evaluated as a ?? (b ?? c). In general terms, an expression of the form E1 ?? E2 ?? ... ?? EN returns the first of the operands that is non-null, or null if all operands are null. *end example*]

The type of the expression a ?? b depends on which implicit conversions are available on the operands. In order of preference, the type of a ?? b is A<sub>0</sub>, A, or B, where A is the type of a (provided that a has a type), B is the type of b (provided that b has a type), and A<sub>0</sub> is the underlying type of A if A is a nullable value type, or A otherwise. Specifically, a ?? b is processed as follows:

- If A exists and is not a nullable value type or a reference type, a compile-time error occurs.
- If b is a dynamic expression, the result type is dynamic. At run-time, a is first evaluated. If a is not null, a is converted to dynamic, and this becomes the result. Otherwise, b is evaluated, and this becomes the result.
- Otherwise, if A exists and is a nullable value type and an implicit conversion exists from b to A<sub>0</sub>, the result type is A<sub>0</sub>. At run-time, a is first evaluated. If a is not null, a is unwrapped to type A<sub>0</sub>, and this becomes the result. Otherwise, b is evaluated and converted to type A<sub>0</sub>, and this becomes the result.
- Otherwise, if A exists and an implicit conversion exists from b to A, the result type is A. At run-time, a is first evaluated. If a is not null, a becomes the result. Otherwise, b is evaluated and converted to type A, and this becomes the result.
- Otherwise, if A exists and is a nullable value type, b has a type B and an implicit conversion exists from A<sub>0</sub> to B, the result type is B. At run-time, a is first evaluated. If a is not null, a is unwrapped to type A<sub>0</sub> and converted to type B, and this becomes the result. Otherwise, b is evaluated and becomes the result.
- Otherwise, if b has a type B and an implicit conversion exists from a to B, the result type is B. At runtime, a is first evaluated. If a is not null, a is converted to type B, and this becomes the result. Otherwise, b is evaluated and becomes the result.

Otherwise, a and b are incompatible, and a compile-time error occurs.

# 12.15 Conditional operator

The **?:** operator is called the conditional operator. It is at times also called the ternary operator.

conditional-expression: null-coalescing-expression null-coalescing-expression ? expression : expression

A conditional expression of the form b ? x : y first evaluates the condition b. Then, if b is true, x is evaluated and becomes the result of the operation. Otherwise, y is evaluated and becomes the result of the operation. A conditional expression never evaluates both x and y.

The conditional operator is right-associative, meaning that operations are grouped from right to left. [*Example*: An expression of the form a ? b : c ? d : e is evaluated as a ? b : (c ? d : e). *end example*]

The first operand of the **?:** operator shall be an expression that can be implicitly converted to bool, or an expression of a type that implements operator true. If neither of these requirements is satisfied, a compile-time error occurs.

The second and third operands, x and y, of the ?: operator control the type of the conditional expression.

- If x has type X and y has type Y then,
  - If X and Y are the same type, then this is the type of the conditional expression.
  - Otherwise, if an implicit conversion (§11.2) exists from X to Y, but not from Y to X, then Y is the type of the conditional expression.
  - Otherwise, if an implicit enumeration conversion (§11.2.4) exists from X to Y, then Y is the type of the conditional expression.
  - Otherwise, if an implicit enumeration conversion (§11.2.4) exists from Y to X, then X is the type of the conditional expression.
  - Otherwise, if an implicit conversion (§11.2) exists from Y to X, but not from X to Y, then X is the type of the conditional expression.
  - Otherwise, no expression type can be determined, and a compile-time error occurs.
- If only one of x and y has a type, and both x and y are implicitly convertible to that type, then that is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

The run-time processing of a conditional expression of the form b ? x : y consists of the following steps:

- First, b is evaluated, and the bool value of b is determined:
  - If an implicit conversion from the type of b to bool exists, then this implicit conversion is performed to produce a bool value.
  - Otherwise, the operator true defined by the type of b is invoked to produce a bool value.
- If the bool value produced by the step above is true, then x is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, y is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

## **12.16 Anonymous function expressions**

#### 12.16.1 General

An *anonymous function* is an expression that represents an "in-line" method definition. An anonymous function does not have a value or type in and of itself, but is convertible to a compatible delegate or

expression-tree type. The evaluation of an anonymous-function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method that the anonymous function defines. If it is an expression-tree type, the conversion evaluates to an expression tree that represents the structure of the method as an object structure.

[*Note*: For historical reasons, there are two syntactic flavors of anonymous functions, namely *lambda-expressions* and *anonymous-method-expressions*. For almost all purposes, *lambda-expressions* are more concise and expressive than *anonymous-method-expressions*, which remain in the language for backwards compatibility. *end note*]

```
lambda-expression:
   async<sub>opt</sub> anonymous-function-signature => anonymous-function-body
anonymous-method-expression:
   async<sub>opt</sub> delegate explicit-anonymous-function-signature<sub>opt</sub> block
anonymous-function-signature:
   explicit-anonymous-function-signature
   implicit-anonymous-function-signature
explicit-anonymous-function-signature:
    ( explicit-anonymous-function-parameter-listopt )
explicit-anonymous-function-parameter-list:
   explicit-anonymous-function-parameter
   explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter
explicit-anonymous-function-parameter:
   anonymous-function-parameter-modifier<sub>opt</sub> type identifier
anonymous-function-parameter-modifier:
    ref
   out
implicit-anonymous-function-signature:
    (implicit-anonymous-function-parameter-list_{opt})
   implicit-anonymous-function-parameter
implicit-anonymous-function-parameter-list:
   implicit-anonymous-function-parameter
   implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter
implicit-anonymous-function-parameter:
   identifier
anonymous-function-body:
   expression
   block
```

The => operator has the same precedence as assignment (=) and is right-associative.

An anonymous function with the async modifier is an async function and follows the rules described in §15.15.

The parameters of an anonymous function in the form of a *lambda-expression* can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types (§11.7).

In a *lambda-expression* with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

( param ) => expr

can be abbreviated to

param => expr

The parameter list of an anonymous function in the form of an *anonymous-method-expression* is optional. If given, the parameters shall be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing out parameters.

A *block* body of an anonymous function is always reachable (§13.2).

[Example: Some examples of anonymous functions follow below:

 $x \Rightarrow x + 1$ // Implicitly typed, expression body // Implicitly typed, statement body  $x => \{ return x + 1; \}$ (int x) => x + 1// Explicitly typed, expression body (int x) => { return x + 1; } // Explicitly typed, statement body  $(x, y) \Rightarrow x * y$ // Multiple parameters () => Console.WriteLine() // No parameters async (t1,t2) => await t1 + await t2 // Async delegate (int x) { return x + 1; } // Anonymous method expression delegate { return 1 + 1; } // Parameter list omitted

#### end example]

The behavior of *lambda-expressions* and *anonymous-method-expressions* is the same except for the following points:

- *anonymous-method-expressions* permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of value parameters.
- *lambda-expressions* permit parameter types to be omitted and inferred whereas *anonymous-method-expressions* require parameter types to be explicitly stated.
- The body of a *lambda-expression* can be an expression or a statement block whereas the body of an *anonymous-method-expression* shall be a statement block.
- Only lambda-expressions have conversions to compatible expression tree types (§9.6).

## 12.16.2 Anonymous function signatures

The *anonymous-function-signature* of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the *anonymous-function-body* (§8.7). Together with the parameter list (if given) the anonymous-method-body constitutes a declaration space (§8.3). It is thus a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant or parameter whose scope includes the *anonymous-method-expression* or *lambda-expression*.

If an anonymous function has an *explicit-anonymous-function-signature*, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order (§11.7). In contrast to method group conversions (§11.8), contra-variance of anonymous function parameter types is not supported. If an anonymous function does not have an *anonymous-function-signature*, then the set of compatible delegate types and expression tree types is restricted to those that have no out parameters.

Note that an *anonymous-function-signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous-function-signature* may be compatible with a delegate type whose parameter list contains a parameter array.

Note also that conversion to an expression tree type, even if compatible, may still fail at compile-time (§9.6).

# 12.16.3 Anonymous function bodies

The body (*expression* or *block*) of an anonymous function is subject to the following rules:

- If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature it can be converted to a delegate type or expression type having parameters (§11.7), but the parameters cannot be accessed in the body.
- Except for ref or out parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a ref or out parameter.
- When the type of this is a struct type, it is a compile-time error for the body to access this. This is true whether the access is explicit (as in this.x) or implicit (as in x where x is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
- The body has access to the outer variables (§12.16.6) of the anonymous function. Access of an outer variable will reference the instance of the variable that is active at the time the *lambda-expression* or *anonymous-method-expression* is evaluated (§12.16.7).
- It is a compile-time error for the body to contain a goto statement, a break statement, or a continue statement whose target is outside the body or within the body of a contained anonymous function.
- A return statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member.

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the *lambda-expression* or *anonymous-method-expression*. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements shall be of a form reserved for compiler use (§7.4.3).

# 12.16.4 Overload resolution

Anonymous functions in an argument list participate in type inference and overload resolution. Refer to §12.6.3 and §12.6.4 for the exact rules.

[Example: The following example illustrates the effect of anonymous functions on overload resolution.

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

The ItemList<T> class has two Sum methods. Each takes a selector argument, which extracts the value to sum over from a list item. The extracted value can be either an int or a double and the resulting sum is likewise either an int or a double.

The Sum methods could for example be used to compute sums from a list of detail lines in an order.

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}
void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

In the first invocation of orderDetails.Sum, both Sum methods are applicable because the anonymous function d => d.UnitCount is compatible with both Func<Detail,int> and Func<Detail,double>. However, overload resolution picks the first Sum method because the conversion to Func<Detail,int> is better than the conversion to Func<Detail,double>.

In the second invocation of orderDetails.Sum, only the second Sum method is applicable because the anonymous function d => d.UnitPrice \* d.UnitCount produces a value of type double.Thus, overload resolution picks the second Sum method for that invocation. *end example*]

# 12.16.5 Anonymous functions and dynamic binding

An anonymous function cannot be a receiver, argument, or operand of a dynamically bound operation.

## 12.16.6 Outer variables

#### 12.16.6.1 General

Any local variable, value parameter, or parameter array whose scope includes the *lambda-expression* or *anonymous-method-expression* is called an *outer variable* of the anonymous function. In an instance function member of a class, the this value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

#### 12.16.6.2 Captured outer variables

When an outer variable is referenced by an anonymous function, the outer variable is said to have been *captured* by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated (§10.2.8). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

[Example: In the example

```
using System;
delegate int D();
class Test
{
   static D F() {
      int x = 0;
      D result = () => ++x;
      return result;
   }
```

```
static void Main() {
    D d = F();
    Console.WriteLine(d());
    Console.WriteLine(d());
    Console.WriteLine(d());
  }
}
```

the local variable x is captured by the anonymous function, and the lifetime of x is extended at least until the delegate returned from F becomes eligible for garbage collection. Since each invocation of the anonymous function operates on the same instance of x, the output of the example is:



#### end example]

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable (§23.4), but is instead considered to be a moveable variable. However, captured outer variables cannot be used in a fixed statement (§23.7), so the address of a captured outer variable cannot be taken.

[*Note*: Unlike an uncaptured variable, a captured local variable can be simultaneously exposed to multiple threads of execution. *end note*]

## 12.16.6.3 Instantiation of local variables

A local variable is considered to be *instantiated* when execution enters the scope of the variable. [*Example*: For example, when the following method is invoked, the local variable x is instantiated and initialized three times—once for each iteration of the loop.

```
static void F() {
   for (int i = 0; i < 3; i++) {
      int x = i * 2 + 1;
      ...
   }
}</pre>
```

However, moving the declaration of x outside the loop results in a single instantiation of x:

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}</pre>
```

#### end example]

When not captured, there is no way to observe exactly how often a local variable is instantiated—because the lifetimes of the instantiations are disjoint, it is possible for each instantiation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

[Example: The example

using System; delegate void D();

```
class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }
    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produces the output:

1 3 5

However, when the declaration of x is moved outside the loop:

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

the output is:

5 5 5

Note that the compiler is permitted (but not required) to optimize the three instantiations into a single delegate instance (§11.7.2).

end example]

If a for-loop declares an iteration variable, that variable itself is considered to be declared outside of the loop. [*Example*: Thus, if the example is changed to capture the iteration variable itself:

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

only one instance of the iteration variable is captured, which produces the output:

3 3 3

# end example]

It is possible for anonymous function delegates to share some captured variables yet have separate instances of others. [*Example*: For example, if F is changed to

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    };
    return result;
}
```

the three delegates capture the same instance of x but separate instances of y, and the output is:

# end example]

Separate anonymous functions can capture the same instance of an outer variable. [*Example*: In the example:

```
using System;
delegate void Setter(int value);
delegate int Getter();
class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        S(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

the two anonymous functions capture the same instance of the local variable x, and they can thus "communicate" through that variable. The output of the example is:

5 10

# end example]

# 12.16.7 Evaluation of anonymous function expressions

An anonymous function F shall always be converted to a delegate type D or an expression-tree type E, either directly or through the execution of a delegate creation expression new D(F). This conversion determines the result of the anonymous function, as described in §11.7.

# 12.16.8 Implementation Exmple

# This subclause is informative.

This subclause describes a possible implementation of anonymous function conversions in terms of other C# constructs. The implementation described here is based on the same principles used by a commercial C# compiler, but it is by no means a mandated implementation, nor is it the only one possible. It only briefly mentions conversions to expression trees, as their exact semantics are outside the scope of this specification.

The remainder of this subclause gives several examples of code that contains anonymous functions with different characteristics. For each example, a corresponding translation to code that uses only other C# constructs is provided. In the examples, the identifier D is assumed by represent the following delegate type:

# public delegate void D();

The simplest form of an anonymous function is one that captures no outer variables:

```
class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}
```

This can be translated to a delegate instantiation that references a compiler generated static method in which the code of the anonymous function is placed:

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }
    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

In the following example, the anonymous function references instance members of this:

```
class Test
{
    int x;
    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

This can be translated to a compiler generated instance method containing the code of the anonymous function:

```
class Test
{
    int x;
    void F() {
        D d = new D(__Method1);
    }
    void __Method1() {
        Console.WriteLine(x);
    }
}
```

In this example, the anonymous function captures a local variable:

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

The lifetime of the local variable must now be extended to at least the lifetime of the anonymous function delegate. This can be achieved by "hoisting" the local variable into a field of a compiler-generated class. Instantiation of the local variable (§12.16.6.3) then corresponds to creating an instance of the compiler generated class, and accessing the local variable corresponds to accessing a field in the instance of the compiler generated class. Furthermore, the anonymous function becomes an instance method of the compiler-generated class:

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }
    class __Locals1
    {
        public int y;
        public void __Method1() {
            Console.writeLine(y);
        }
    }
}
```

Finally, the following anonymous function captures this as well as two local variables with different lifetimes:

```
class Test
{
    int x;
    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        };
    }
}
```

Here, a compiler-generated class is created for each statement block in which locals are captured such that the locals in the different blocks can have independent lifetimes. An instance of \_\_Locals2, the compiler generated class for the inner statement block, contains the local variable z and a field that references an instance of \_\_Locals1. An instance of \_\_Locals1, the compiler generated class for the outer statement block, contains the local variable z and a field that references an instance of \_\_Locals1. An instance of \_\_Locals1, the compiler generated class for the outer statement block, contains the local variable y and a field that references this of the enclosing function member. With these data structures, it is possible to reach all captured outer variables through an instance of \_\_Local2, and the code of the anonymous function can thus be implemented as an instance method of that class.

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
        D d = new D(__locals2.._Method1);
    }
    class __Locals1
    {
        public Test __this;
        public int y;
    }
     class __Locals2
        {
            public __Locals1 __locals1;
            public int z;
        }
        class1;
        __locals2.__locals1 __locals1;
        public int z;
        }
    }
}
</pre>
```

```
public void __Method1() {
    Console.WriteLine(__locals1.__this.x + __locals1.y + z);
    }
}
```

The same technique applied here to capture local variables can also be used when converting anonymous functions to expression trees: references to the compiler-generated objects can be stored in the expression tree, and access to the local variables can be represented as field accesses on these objects. The advantage of this approach is that it allows the "lifted" local variables to be shared between delegates and expression trees.

End of informative text.

# 12.17 Query expressions

# 12.17.1 General

*Query expressions* provide a language-integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

```
query-expression:
    from-clause query-body
from-clause:
    from type<sub>opt</sub> identifier in expression
query-body:
    query-body-clauses<sub>opt</sub> select-or-group-clause query-continuation<sub>opt</sub>
query-body-clauses:
    query-body-clause
    query-body-clauses query-body-clause
query-body-clause:
    from-clause
    let-clause
    where-clause
    join-clause
    join-into-clause
    orderby-clause
let-clause:
    let identifier = expression
where-clause:
    where boolean-expression
join-clause:
    join type<sub>opt</sub> identifier in expression on expression equals expression
join-into-clause:
    join typeopt identifier in expression on expression equals expression into
    identifier
orderby-clause:
    orderby orderings
orderings:
    ordering
    orderings, ordering
ordering:
    expression ordering-direction<sub>opt</sub>
```

ordering-direction: ascending descending select-or-group-clause: select-clause group-clause select-clause: select expression group-clause: group expression by expression query-continuation: into identifier query-body

A query expression begins with a from clause and ends with either a select or group clause. The initial from clause may be followed by zero or more from, let, where, join or orderby clauses. Each from clause is a generator introducing a *range variable* that ranges over the elements of a *sequence*. Each let clause introduces a range variable representing a value computed by means of previous range variables. Each where clause is a filter that excludes items from the result. Each join clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each orderby clause reorders items according to specified criteria. The final select or group clause specifies the shape of the result in terms of the range variables. Finally, an into clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

# 12.17.2 Ambiguities in query expressions

Query expressions use a number of contextual keywords (§7.4.4): ascending, by, descending, equals, from, group, into, join, let, on, orderly, select and where.

To avoid ambiguities that could arise from the use of these identifiers both as keywords and simple names these identifiers are considered keywords anywhere within a query expression, unless they are prefixed with "@" (§7.4.4) in which case they are considered identifiers. For this purpose, a query expression is any expression that starts with "from *identifier*" followed by any token except ";", "=" or ",".

# 12.17.3 Query expression translation

# 12.17.3.1 General

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the query-expression pattern (§12.17.4). Specifically, query expressions are translated into invocations of methods named Where, Select, SelectMany, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, GroupBy, and Cast. These methods are expected to have particular signatures and return types, as described in §12.17.4. These methods may be instance methods of the object being queried or extension methods that are external to the object. These methods implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover compile time errors. These error conditions include, but are not limited to, methods that do not exist, arguments of the wrong types, and generic methods where type inference fails.

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of application: each section assumes that the translations in the preceding sections have been performed exhaustively, and once exhausted, a section will not later be revisited in the processing of the same query expression.

It is a compile time error for a query expression to include an assignment to a range variable, or the use of a range variable as an argument for a ref or out parameter.

Certain translations inject range variables with *transparent identifiers* denoted by \*. These are described further in §12.17.3.8.

12.17.3.2 select and group ... by clauses with continuations

A query expression with a group clause using a property Prop of y and a query body Q containing a continuation in the form:

from y in S group y by y. Prop into x Q

is translated into:

from x in ( from y in S group y by y.Prop ) Q

The translations in the following sections assume that queries have no into continuations.

[Example: The example:

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

is translated into:

```
from g in
   (from c in customers
   group c by c.Country)
select new { Country = g.Key, CustCount = g.Count() }
```

the final translation of which is:

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

end example]

12.17.3.3 Explicit range variable types

A from clause that explicitly specifies a range variable type

from T x in e

is translated into

from x in (e). Cast < T > ()

A join clause that explicitly specifies a range variable type

join T x in e on  $k_1$  equals  $k_2$ 

is translated into

join x in (e). Cast  $\langle T \rangle$  () on  $k_1$  equals  $k_2$ 

The translations in the following sections assume that queries have no explicit range variable types.

[Example: The example

```
from Customer c in customers
where c.City == "London"
select c
```

is translated into

```
from c in (customers).Cast<Customer>()
where c.City == "London"
select c
```

the final translation of which is

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

end example]

[*Note*: Explicit range variable types are useful for querying collections that implement the non-generic IEnumerable interface, but not the generic IEnumerable<T> interface. In the example above, this would be the case if customers were of type ArrayList. *end note*]

12.17.3.4 Degenerate query expressions

A query expression of the form

from x in e select x

is translated into

(e) . Select  $(x \Rightarrow x)$ 

[*Example*: The example

from c in customers
select c

Is translated into

(customers).Select(c => c)

end example]

A degenerate query expression is one that trivially selects the elements of the source.

[Note: Later phases of the translation (§12.17.3.6 and §12.17.3.7) remove degenerate queries introduced by other translation steps by replacing them with their source. It is important, however, to ensure that the result of a query expression is never the source object itself. Otherwise, returning the result of such a query might inadvertently expose private data (e.g., an element array) to a caller. Therefore this step protects degenerate queries written directly in source code by explicitly calling Select on the source. It is then up to the implementers of Select and other query operators to ensure that these methods never return the source object itself.end note]

12.17.3.5 From, let, where, join and orderby clauses

A query expression with a second from clause followed by a select clause

```
from x_1 in e_1
from x_2 in e_2
select v
```

is translated into

```
(e_1) . SelectMany(x_1 \Rightarrow e_2, (x_1, x_2) \Rightarrow v)
```

[Example: The example

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
(customers).
SelectMany(c => c.Orders,
    (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

end example]

A query expression with a second from clause followed by a query body Q containing a non-empty set of query body clauses:

from  $x_1$  in  $e_1$ from  $x_2$  in  $e_2$ Q

is translated into

```
from * in ( e_1 ) . SelectMany( x_1 \Rightarrow e_2 , ( x_1 , x_2 ) \Rightarrow new { x_1 , x_2 } ) Q
```

[Example: The example

from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }

is translated into

```
from * in (customers).
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

the final translation of which is

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

where x is a compiler generated identifier that is otherwise invisible and inaccessible. *end example*]

A let expression along with its preceding from clause:

from x in elet y = f

is translated into

```
from * in ( e ) . Select ( x \Rightarrow new \{ x , y = f \} )
```

[Example: The example

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

is translated into

```
from * in (orders).
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
})
where t >= 1000
select new { o.OrderID, Total = t }
```

the final translation of which is

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

where x is a compiler generated identifier that is otherwise invisible and inaccessible. *end example*]

A where expression along with its preceding from clause:

```
from x in e
where f
...
```

is translated into

from x in (e). Where  $(x \Rightarrow f)$ 

A join clause immediately followed by a select clause

from  $x_1$  in  $e_1$ join  $x_2$  in  $e_2$  on  $k_1$  equals  $k_2$ select v

is translated into

 $(e_1)$  . Join $(e_2, x_1 \Rightarrow k_1, x_2 \Rightarrow k_2, (x_1, x_2) \Rightarrow v)$ 

[Example: The example

from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }

is translated into

```
(customers).Join(orders, c => c.CustomerID, o => o.CustomerID,
  (c, o) => new { c.Name, o.OrderDate, o.Total })
```

end example]

A join clause followed by a query body clause:

from  $x_1$  in  $e_1$ join  $x_2$  in  $e_2$  on  $k_1$  equals  $k_2$ 

is translated into

from \* in (  $e_1$  ) . Join(  $e_2$  ,  $x_1 \Rightarrow k_1$  ,  $x_2 \Rightarrow k_2$  , (  $x_1$  ,  $x_2$  )  $\Rightarrow$  new {  $x_1$  ,  $x_2$  })

A join-into clause immediately followed by a select clause

from  $x_1$  in  $e_1$ join  $x_2$  in  $e_2$  on  $k_1$  equals  $k_2$  into gselect v

is translated into

( $e_1$ ). GroupJoin( $e_2$ ,  $x_1 \Rightarrow k_1$ ,  $x_2 \Rightarrow k_2$ , ( $x_1$ , g)  $\Rightarrow v$ )

A join into clause followed by a query body clause

from  $x_1$  in  $e_1$ join  $x_2$  in  $e_2$  on  $k_1$  equals  $k_2$  into g

... is translated into

> from \* in ( $e_1$ ). GroupJoin(  $e_2$ ,  $x_1 \Rightarrow k_1$ ,  $x_2 \Rightarrow k_2$ , ( $x_1$ , g)  $\Rightarrow$  new { $x_1$ , g})

[Example: The example

from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }

is translated into

```
from * in (customers).
   GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

the final translation of which is

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

where x and y are compiler generated identifiers that are otherwise invisible and inaccessible. *end example*]

An orderby clause and its preceding from clause:

from x in e orderby  $k_1$ ,  $k_2$ , ...,  $k_n$ ...

is translated into

from x in ( e ) . OrderBy (  $x \Rightarrow k_1$  ) . ThenBy (  $x \Rightarrow k_2$  ) . ..... ThenBy (  $x \Rightarrow k_n$  )

If an ordering clause specifies a descending direction indicator, an invocation of OrderByDescending or ThenByDescending is produced instead.

[Example: The example

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

has the final translation

```
(orders).
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

#### end example]

The following translations assume that there are no let, where, join or orderby clauses, and no more than the one initial from clause in each query expression.

#### 12.17.3.6 Select clauses

A query expression of the form

from x in e select v

is translated into

(e) . Select  $(x \Rightarrow v)$ 

except when v is the identifier x, the translation is simply

( e )

[Example: The example

```
from c in customers.where(c => c.City == "London")
select c
```

is simply translated into

```
(customers).where(c => c.City == "London")
```

end example]

12.17.3.7 Group clauses

A group clause

from x in e group v by k

is translated into

(e). GroupBy  $(x \Rightarrow k, x \Rightarrow v)$ 

except when v is the identifier x, the translation is

(e) . GroupBy  $(x \Rightarrow k)$ 

[Example: The example

from c in customers
group c.Name by c.Country

is translated into

```
(customers).
GroupBy(c => c.Country, c => c.Name)
```

end example]

12.17.3.8 Transparent identifiers

Certain translations inject range variables with *transparent identifiers* denoted by \*. Transparent identifiers exist only as an intermediate step in the query-expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

- When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.
- When a member with a transparent identifier is in scope, the members of that member are in scope as well.
- When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.

In the translation steps described above, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and shows one possible translation of transparent identifiers.

[Example: The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

is translated into

```
from * in (customers).
   SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

which is further translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

which, when transparent identifiers are erased, is equivalent to

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

where x is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

is translated into

```
from * in (customers).
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

which is further reduced to

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o
}).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p
}).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

the final translation of which is

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
  (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
  (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
  (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

where x, y, and z are compiler-generated identifiers that are otherwise invisible and inaccessible.

end example]

## 12.17.4 The query-expression pattern

The **Query-expression pattern** establishes a pattern of methods that types can implement to support query expressions.

A generic type C<T> supports the query-expression-pattern if its public member methods and the publicly accessible extension methods could be replaced by the following class definition. The members and accessible extension methods is referred to as the "shape" of a generic type C<T>. A generic type is used in order to illustrate the proper relationships between parameter and return types, but it is possible to implement the pattern for non-generic types as well.

```
delegate R Func<T1,R>(T1 arg1);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
class C
{
   public C<T> Cast<T>();
}
class C<T> : C
   public C<T> where(Func<T, bool> predicate);
   public C<U> Select<U>(Func<T,U> selector);
   public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
      Func<T,U,V> resultSelector);
   public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
      Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);
   public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
      Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);
   public O<T> OrderBy<K>(Func<T,K> keySelector);
   public O<T> OrderByDescending<K>(Func<T,K> keySelector);
   public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
   public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
      Func<T.E> elementSelector);
}
class O<T> : C<T>
   public O<T> ThenBy<K>(Func<T,K> keySelector);
   public 0<T> ThenByDescending<K>(Func<T,K> keySelector);
}
class G<K,T> : C<T>
{
   public K Key { get; }
}
```

The methods above use the generic delegate types Func<T1, R> and Func<T1, T2, R>, but they could equally well have used other delegate or expression-tree types with the same relationships in parameter and return types.

[*Note*: The recommended relationship between C<T> and O<T> that ensures that the ThenBy and ThenByDescending methods are available only on the result of an OrderBy or OrderByDescending. *end note*]

[*Note*: The recommended shape of the result of GroupBy—a sequence of sequences, where each inner sequence has an additional Key property. *end note*]

[*Note*: Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement any or all of the query-expression pattern. For example, the methods of the pattern can be implemented as instance methods or as extension methods

because the two have the same invocation syntax, and the methods can request delegates or expression trees because anonymous functions are convertible to both. Types implementing only some of the query expression pattern support only query expression translations that map to the methods that type supports. *end note*]

[*Note*: The System.Linq namespace provides an implementation of the query-expression pattern for any type that implements the System.Collections.Generic.IEnumerable<T> interface. end note]

# 12.18 Assignment operators

# 12.18.1 General

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

assignment: unary-expression assignment-operator expression assignment-operator: = += -=

-= \*= /= %= &= |= ^= <<= right-shift-assignment

The left operand of an assignment shall be an expression classified as a variable, a property access, an indexer access, or an event access.

The = operator is called the *simple assignment operator*. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator shall not be an event access (except as described in §15.8.2). The simple assignment operator is described in §12.18.2.

The assignment operators other than the = operator are called the *compound assignment operators*. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in §12.18.3.

The += and -= operators with an event access expression as the left operand are called the *event assignment operators*. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in §12.18.4.

The assignment operators are right-associative, meaning that operations are grouped from right to left. [*Example*: An expression of the form a = b = c is evaluated as a = (b = c). *end example*]

# 12.18.2 Simple assignment

The = operator is called the simple assignment operator.

If the left operand of a simple assignment is of the form E.P or E[Ei] where E has the compile-time type dynamic, then the assignment is dynamically bound (§12.3.3). In this case, the compile-time type of the assignment expression is dynamic, and the resolution described below will take place at run-time based on the run-time type of E. If the left operand is of the form E[Ei] where at least one element of Ei has the compile-time type dynamic, and the compile-time type of E is not an array, the resulting indexer access is dynamically bound, but with limited compile-time checking (§12.6.5).

In a simple assignment, the right operand shall be an expression that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand, and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer shall have an accessible set accessor. If this is not the case, a binding-time error occurs.

The run-time processing of a simple assignment of the form x = y consists of the following steps:

- If x is classified as a variable:
  - x is evaluated to produce the variable.
  - y is evaluated and, if required, converted to the type of x through an implicit conversion (§11.2).
  - If the variable given by x is an array element of a *reference-type*, a run-time check is performed to ensure that the value computed for y is compatible with the array instance of which x is an element. The check succeeds if y is null, or if an implicit reference conversion (§11.2.7) exists from the -type of the instance referenced by y to the actual element type of the array instance containing x. Otherwise, a System.ArrayTypeMismatchException is thrown.
  - The value resulting from the evaluation and conversion of y is stored into the location given by the evaluation of x.
- If x is classified as a property or indexer access:
  - The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent set accessor invocation.
  - y is evaluated and, if required, converted to the type of x through an implicit conversion (§11.2).
  - The set accessor of x is invoked with the value computed for y as its value argument.

[*Note*: if the compile time type of x is dynamic and there is an implicit conversion from the compile time type of y to dynamic, no runtime resolution is required. *end note*]

[*Note*: The array co-variance rules (§17.6) permit a value of an array type A[] to be a reference to an instance of an array type B[], provided an implicit reference conversion exists from B to A. Because of these rules, assignment to an array element of a *reference-type* requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

the last assignment causes a System.ArrayTypeMismatchException to be thrown because a reference to an ArrayList cannot be stored in an element of a string[]. end note]

When a property or indexer declared in a *struct-type* is the target of an assignment, the instance expression associated with the property or indexer access shall be classified as a variable. If the instance expression is classified as a value, a binding-time error occurs. [*Note*: Because of §12.7.5, the same rule also applies to fields. *end note*]

[*Example*: Given the declarations:

struct Point
{
 int x, y;

```
public Point(int x, int y) {
       this.x = x;
       this.y = y;
   }
    public int X {
       get { return x; }
set { x = value; }
   }
   public int Y {
   get { return y; }
   set { y = value; }
   }
}
struct Rectangle
   Point a, b;
   public Rectangle(Point a, Point b) {
       this.a = a;
       this.b = b;
   }
   public Point A {
       get { return a; }
        set { a = value; }
   }
   public Point B {
       get { return b; }
set { b = value; }
   }
}
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to p.X, p.Y, r.A, and r.B are permitted because p and r are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid, since r.A and r.B are not variables. end example]

# 12.18.3 Compound assignment

If the left operand of a compound assignment is of the form E.P or E[Ei] where E has the compile-time type dynamic, then the assignment is dynamically bound (§12.3.3). In this case, the compile-time type of the assignment expression is dynamic, and the resolution described below will take place at run-time based on the run-time type of E. If the left operand is of the form E[Ei] where at least one element of Ei has the compile-time type dynamic, and the compile-time type of E is not an array, the resulting indexer access is dynamically bound, but with limited compile-time checking (§12.6.5).

An operation of the form x op= y is processed by applying binary operator overload resolution (§12.4.5) as if the operation was written x op y. Then,

• If the return type of the selected operator is implicitly convertible to the type of x, the operation is evaluated as x = x op y, except that x is evaluated only once.

- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is explicitly convertible to the type of x, and if y is implicitly convertible to the type of x or the operator is a shift operator, then the operation is evaluated as x = (T) (x op y), where T is the type of x, except that x is evaluated only once.
- Otherwise, the compound assignment is invalid, and a binding-time error occurs.

The term "evaluated only once" means that in the evaluation of x op y, the results of any constituent expressions of x are temporarily saved and then reused when performing the assignment to x. [*Example*: In the assignment A() [B()] += C(), where A is a method returning int[], and B and C are methods returning int, the methods are invoked only once, in the order A, B, C. end example]

When the left operand of a compound assignment is a property access or indexer access, the property or indexer shall have both a get accessor and a set accessor. If this is not the case, a binding-time error occurs.

The second rule above permits x op= y to be evaluated as x = (T) (x op y) in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type sbyte, byte, short, ushort, or char. Even when both arguments are of one of those types, the predefined operators produce a result of type int, as described in §12.4.7.3. Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that x op = y is permitted if both of x op y and x = y are permitted. [*Example*: In the following code

the intuitive reason for each error is that a corresponding simple assignment would also have been an error. *end example*]

[*Note*: This also means that compound assignment operations support lifted operators. Since a compound assignment x op= y is evaluated as either x = x op y or x = (T) (x op y), the rules of evaluation implicitly cover lifted operators. *end note*]

# 12.18.4 Event assignment

If the left operand of a += or -= operator is classified as an event access, then the expression is evaluated as follows:

- The instance expression, if any, of the event access is evaluated.
- The right operand of the += or -= operator is evaluated, and, if required, converted to the type of the left operand through an implicit conversion (§11.2).
- An event accessor of the event is invoked, with an argument list consisting of the value computed in the previous step. If the operator was +=, the add accessor is invoked; if the operator was -=, the remove accessor is invoked.

An event assignment expression does not yield a value. Thus, an event assignment expression is valid only in the context of a *statement-expression* (§13.7).

# 12.19 Expression

An expression is either a non-assignment-expression or an assignment.

expression: non-assignment-expression assignment

non-assignment-expression: conditional-expression lambda-expression query-expression

# 12.20 Constant expressions

A constant expression is an expression that shall be fully evaluated at compile-time.

constant-expression: expression

A constant expression may be either a value type or a reference type. If a constant expression is a value type, it must be one of the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, or any enumeration type. If a constant expression is a reference type, it must be the string type, a default value expression (§12.7.15) for some reference type, or the value of the expression must be null.

Only the following constructs are permitted in constant expressions:

- Literals (including the null literal).
- References to const members of class and struct types.
- References to members of enumeration types.
- References to const parameters or local variables
- Parenthesized subexpressions, which are themselves constant expressions.
- Cast expressions.
- checked and unchecked expressions
- The predefined +, -, !, and ~ unary operators.
- The predefined +, -, \*, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <=, and >= binary operators.
- The ?: conditional operator.
- sizeof expressions, provided the unmanaged-type is one of the types specified in §23.6.9 for which sizeof returns a constant value.
- Default value expressions, provided the type is one of the types listed above.

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided the source of the conversions is a constant expression that evaluates to the null value.

[*Note*: Other conversions including boxing, unboxing, and implicit reference conversions of non-null values are not permitted in constant expressions. *end note*]

[Example: In the following code

```
class C {
    const object i = 5; // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of i is an error because a boxing conversion is required. The initialization of str is an error because an implicit reference conversion from a non-null value is required. *end example*]

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a subexpression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of nonconstant expressions, except that where run-time evaluation would have thrown an exception, compiletime evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an unchecked context, overflows that occur in integraltype arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (§12.7.14).

Constant expressions are required in the contexts listed below and this is indicated in the grammar by using *constant-expression*. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations (§15.4)
- Enumeration member declarations (§19.4)
- Default arguments of formal parameter lists (§15.6.2)
- case labels of a switch statement (§13.8.3).
- goto case statements (§13.10.4)
- Dimension lengths in an array creation expression (§12.7.11.5) that includes an initializer.
- Attributes (§22)

An implicit constant expression conversion (§11.2.10) permits a constant expression of type int to be converted to sbyte, byte, short, ushort, uint, or ulong, provided the value of the constant expression is within the range of the destination type.

# 12.21 Boolean expressions

A *boolean-expression* is an expression that yields a result of type bool; either directly or through application of operator true in certain contexts as specified in the following:

boolean-expression: expression

The controlling conditional expression of an *if-statement* (§13.8.2), *while-statement* (§13.9.2), *do-statement* (§13.9.3), or *for-statement* (§13.9.4) is a *boolean-expression*. The controlling conditional expression of the ?: operator (§12.15) follows the same rules as a *boolean-expression*, but for reasons of operator precedence is classified as a *conditional-or-expression*.

A *boolean-expression* E is required to be able to produce a value of type bool, as follows:

- If E is implicitly convertible to bool then at run-time that implicit conversion is applied.
- Otherwise, unary operator overload resolution (§12.4.4) is used to find a unique best implementation of operator true on E, and that implementation is applied at run-time.
- If no such operator is found, a binding-time error occurs.

# 13. Statements

# 13.1 General

C# provides a variety of statements. [*Note*: Most of these statements will be familiar to developers who have programmed in C and C++. *end note*]

statement: labeled-statement declaration-statement embedded-statement

embedded-statement: block empty-statement expression-statement iteration-statement jump-statement try-statement checked-statement unchecked-statement using-statement yield-statement

The *embedded-statement* nonterminal is used for statements that appear within other statements. The use of *embedded-statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. [*Example*: The code

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

results in a compile-time error because an if statement requires an *embedded-statement* rather than a *statement* for its if branch. If this code were permitted, then the variable i would be declared, but it could never be used. Note, however, that by placing i's declaration in a block, the example is valid. *end example*]

# 13.2 End points and reachability

Every statement has an *end point*. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. [*Example*: When control reaches the end point of a statement in a block, control is transferred to the next statement in the block. *end example*]

If a statement can possibly be reached by execution, the statement is said to be *reachable*. Conversely, if there is no possibility that a statement will be executed, the statement is said to be *unreachable*.

[Example: In the following code

```
void F() {
   Console.WriteLine("reachable");
   goto Label;
   Console.WriteLine("unreachable");
   Label:
   Console.WriteLine("reachable");
}
```

the second invocation of Console.WriteLine is unreachable because there is no possibility that the statement will be executed. *end example*]

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

[*Note*: To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (§12.20) that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

the Boolean expression of the if statement is a constant expression because both operands of the == operator are constants. As the constant expression is evaluated at compile-time, producing the value false, the Console.WriteLine invocation is considered unreachable. However, if i is changed to be a local variable

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

the Console.WriteLine invocation is considered reachable, even though, in reality, it will never be executed. *end note*]

The *block* of a function member or an anonymous function is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

[Example: In the following code

```
void F(int x) {
   Console.WriteLine("start");
   if (x < 0) Console.WriteLine("negative");
}</pre>
```

the reachability of the second Console.WriteLine is determined as follows:

- The first Console.WriteLine expression statement is reachable because the block of the F method is reachable (§13.3).
- The end point of the first Console.WriteLine expression statement is reachable because that statement is reachable (§13.7 and §13.3).
- The if statement is reachable because the end point of the first Console.WriteLine expression statement is reachable (§13.7 and §13.3).
- The second Console.WriteLine expression statement is reachable because the Boolean expression of the if statement does not have the constant value false.

end example]

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the switch statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a break statement is missing.
- It is a compile-time error for the end point of the block of a function member or an anonymous function that computes a value to be reachable. If this error occurs, it typically is an indication that a return statement is missing (§13.10.5).

# 13.3 Blocks

# 13.3.1 General

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

block:

{ statement-list<sub>opt</sub> }

A *block* consists of an optional *statement-list* (§13.3.2), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements (§13.6). The scope of a local variable or constant declared in a block is the block.

Within a block, the meaning of a name used in an expression context shall always be the same (§12.7.3.2).

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A *block* that contains one or more yield statements (§13.15) is called an iterator block. Iterator blocks are used to implement function members as iterators (§15.14). Some additional restrictions apply to iterator blocks:

- It is a compile-time error for a return statement to appear in an iterator block (but yield return statements are permitted).
- It is a compile-time error for an iterator block to contain an unsafe context (§23.2). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

#### 13.3.2 Statement lists

A *statement list* consists of one or more statements written in sequence. Statement lists occur in *blocks* (§13.3) and in *switch-blocks* (§13.8.3).

statement-list: statement statement-list statement

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable goto statement.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

# 13.4 The empty statement

An empty-statement does nothing.

empty-statement:

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

[*Example*: An empty statement can be used when writing a while statement with a null body:

```
bool ProcessMessage() {...}
void ProcessMessages() {
    while (ProcessMessage())
    ;
}
```

Also, an empty statement can be used to declare a label just before the closing "}" of a block:

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

end example]

# 13.5 Labeled statements

A *labeled-statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

labeled-statement: identifier : statement

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from goto statements (§13.10.4) within the scope of the label. [*Note*: This means that goto statements can transfer control within blocks and out of blocks, but never into blocks. *end note*]

Labels have their own declaration space and do not interfere with other identifiers. [Example: The example

```
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

is valid and uses the name x as both a parameter and a label. end example]

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable goto statement, unless the goto statement is inside the try block or a catch block of a *try-statement* that includes a finally block whose end point is unreachable, and the labeled statement is outside the *try-statement*.

# **13.6 Declaration statements**

# 13.6.1 General

A *declaration-statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

declaration-statement: local-variable-declaration ; local-constant-declaration ;

#### 13.6.2 Local variable declarations

A local-variable-declaration declares one or more local variables.

local-variable-declaration: local-variable-type local-variable-declarators

local-variable-type: type var

local-variable-declarators: local-variable-declarator local-variable-declarators , local-variable-declarator

local-variable-declarator: identifier

identifier = local-variable-initializer

local-variable-initializer: expression array-initializer

The *local-variable-type* of a *local-variable-declaration* either directly specifies the type of the variables introduced by the declaration, or indicates with the identifier var that the type should be inferred based on an initializer. The type is followed by a list of *local-variable-declarators*, each of which introduces a new variable. A *local-variable-declarator* consists of an *identifier* that names the variable, optionally followed by an "=" token and a *local-variable-initializer* that gives the initial value of the variable.

In the context of a local variable declaration, the identifier var acts as a contextual keyword (§7.4.4). When the *local-variable-type* is specified as var and no type named var is in scope, the declaration is an *implicitly typed local variable declaration*, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

- The local-variable-declaration cannot include multiple local-variable-declarators.
- The *local-variable-declarator* shall include a *local-variable-initializer*.
- The *local-variable-initializer* shall be an *expression*.
- The initializer *expression* shall have a compile-time type.
- The initializer expression cannot refer to the declared variable itself

[*Example*: The following are incorrect implicitly typed local variable declarations:

var x;	<pre>// Error, no initializer to infer type from</pre>
var y = {1, 2, 3};	// Error, array initializer not permitted
var z = null;	<pre>// Error, null does not have a type</pre>
var u = x => x + 1;	// Error, anonymous functions do not have a type
var v = v++;	// Error, initializer cannot refer to v itself

end example]

The value of a local variable is obtained in an expression using a *simple-name* (§12.7.3), and the value of a local variable is modified using an *assignment* (§12.18). A local variable shall be definitely assigned (§10.4) at each location where its value is obtained.

The scope of a local variable declared in a *local-variable-declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

[*Example*: The example

void F() {
 int x = 1, y, z = x \* 2;
}

corresponds exactly to

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

end example]

In an implicitly typed local variable declaration, the type of the local variable being declared is taken to be the same as the type of the expression used to initialize the variable. [*Example*:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

end example]

#### 13.6.3 Local constant declarations

A local-constant-declaration declares one or more local constants.

```
local-constant-declaration:
const type constant-declarators
```

constant-declarators: constant-declarator constant-declarators , constant-declarator constant-declarator: identifier = constant-expression

The *type* of a *local-constant-declaration* specifies the type of the constants introduced by the declaration. The type is followed by a list of *constant-declarators*, each of which introduces a new constant. A *constant-declarator* consists of an *identifier* that names the constant, followed by an "=" token, followed by a *constant-expression* (§12.20) that gives the value of the constant.

The *type* and *constant-expression* of a local constant declaration shall follow the same rules as those of a constant member declaration (§15.4).

The value of a local constant is obtained in an expression using a *simple-name* (§12.7.3).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes the end of its *constant-declarator*. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

# **13.7 Expression statements**

An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement: statement-expression ;

statement-expression: invocation-expression object-creation-expression assignment post-increment-expression pre-decrement-expression pre-decrement-expression await-expression

Not all expressions are permitted as statements. [*Note*: In particular, expressions such as x + y and x == 1, that merely compute a value (which will be discarded), are not permitted as statements. *end note*]

Execution of an expression statement evaluates the contained expression and then transfers control to the end point of the expression statement. The end point of an *expression-statement* is reachable if that *expression-statement* is reachable.

#### **13.8 Selection statements**

#### 13.8.1 General

Selection statements select one of a number of possible statements for execution based on the value of some expression.

selection-statement: if-statement switch-statement

#### 13.8.2 The if statement

The if statement selects a statement for execution based on the value of a Boolean expression.

if-statement:

if ( boolean-expression ) embedded-statement

if ( boolean-expression ) embedded-statement else embedded-statement

An else part is associated with the lexically nearest preceding if that is allowed by the syntax. [*Example*: Thus, an if statement of the form

if (x) if (y) F(); else G();

is equivalent to

```
if (x) {
    if (y)
        F();
    }
    else {
        G();
    }
}
```

end example]

An if statement is executed as follows:

- The boolean-expression (§12.21) is evaluated.
- If the Boolean expression yields true, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
- If the Boolean expression yields false and if an else part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
- If the Boolean expression yields false and if an else part is not present, control is transferred to the end point of the if statement.

The first embedded statement of an if statement is reachable if the if statement is reachable and the Boolean expression does not have the constant value false.

The second embedded statement of an if statement, if present, is reachable if the if statement is reachable and the Boolean expression does not have the constant value true.

The end point of an if statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an if statement with no else part is reachable if the if statement is reachable and the Boolean expression does not have the constant value true.

# 13.8.3 The switch statement

The switch statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

```
switch-statement:
    switch ( expression ) switch-block
switch-block:
    { switch-sections<sub>opt</sub> }
switch-sections:
    switch-section
    switch-section
switch-section:
    switch-labels statement-list
```

switch-labels: switch-label switch-labels switch-label switch-label: case constant-expression : default :

A *switch-statement* consists of the keyword *switch*, followed by a parenthesized expression (called the *switch expression*), followed by a *switch-block*. The *switch-block* consists of zero or more *switch-sections*, enclosed in braces. Each *switch-section* consists of one or more *switch-labels* followed by a *statement-list* (§13.3.2).

The governing type of a switch statement is established by the switch expression.

- If the type of the switch expression is sbyte, byte, short, ushort, int, uint, long, ulong, char, bool, string, or an enum-type, or if it is the nullable value type corresponding to one of these types, then that is the governing type of the switch statement.
- Otherwise, exactly one user-defined implicit conversion shall exist from the type of the switch expression to one of the following possible governing types: sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or, a nullable value type corresponding to one of those types.
- Otherwise, a compile-time error occurs.

The constant expression of each case label shall denote a value of a type that is implicitly convertible (§11.2) to the governing type of the switch statement. A compile-time error occurs if two or more case labels in the same switch statement specify the same constant value.

There can be at most one default label in a switch statement.

A switch statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a case label in the same switch statement is equal to the value of the switch expression, control is transferred to the statement list following the matched case label.
- If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression, and if a default label is present, control is transferred to the statement list following the default label.
- If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression, and if no default label is present, control is transferred to the end point of the switch statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. [*Example*: The example

```
switch (i) {
    case 0:
        CaseZero();
        break;
    case 1:
        CaseOne();
        break;
    default:
        CaseOthers();
        break;
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to "fall through" to the next switch section, and the example

```
switch (i) {
    case 0:
        CaseZero();
    case 1:
        CaseZeroOrOne();
    default:
        CaseAny();
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit goto case or goto default statement shall be used:

```
switch (i) {
    case 0:
        CaseZero();
        goto case 1;
    case 1:
        CaseZeroOrOne();
        goto default;
        default:
        CaseAny();
        break;
}
```

end example]

Multiple labels are permitted in a switch-section. [Example: The example

```
switch (i) {
    case 0:
        CaseZero();
        break;
    case 1:
        CaseOne();
        break;
    case 2:
    default:
        CaseTwo();
        break;
}
```

is valid. The example does not violate the "no fall through" rule because the labels case 2: and default: are part of the same *switch-section*. *end example*]

[*Note*: The "no fall through" rule prevents a common class of bugs that occur in C and C++ when break statements are accidentally omitted. For example, the sections of the switch statement above can be reversed without affecting the behavior of the statement:

```
switch (i) {
    default:
        CaseAny();
        break;
    case 1:
        CaseZeroOrOne();
        goto default;
        case 0:
        CaseZero();
        goto case 1;
}
```

```
end note]
```

[Note: The statement list of a switch section typically ends in a break, goto case, or goto default statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a while statement controlled by the Boolean expression true is known to never reach its end point. Likewise, a throw or return statement always transfers control elsewhere and never reaches its end point. Thus, the following example is valid:

```
switch (i) {
    case 0:
        while (true) F();
    case 1:
        throw new ArgumentException();
    case 2:
        return;
}
```

end note]

[*Example*: The governing type of a switch statement can be the type string. For example:

```
void DoCommand(string command) {
   switch (command.ToLower()) {
      case "run":
        DoRun();
        break;
      case "save":
        DoSave();
        break;
      case "quit":
        DoQuit();
        break;
      default:
        InvalidCommand(command);
        break;
   }
}
```

#### end example]

[Note: Like the string equality operators (§12.11.8), the switch statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a case label constant. end note]

When the governing type of a switch statement is string or a nullable value type, the value null is permitted as a case label constant.

The *statement-lists* of a *switch-block* may contain declaration statements (§13.6). The scope of a local variable or constant declared in a switch block is the switch block.

Within a switch block, the meaning of a name used in an expression context shall always be the same (§12.7.3.2).

The statement list of a given switch section is reachable if the switch statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a case label in the switch section.
- The switch expression is a constant value that doesn't match any case label, and the switch section contains the default label.
- A switch label of the switch section is referenced by a reachable goto case or goto default statement.

The end point of a switch statement is reachable if at least one of the following is true:

- The switch statement contains a reachable break statement that exits the switch statement.
- The switch statement is reachable, the switch expression is a non-constant value, and no default label is present.
- The switch statement is reachable, the switch expression is a constant value that doesn't match any case label, and no default label is present.

# 13.9 Iteration statements

#### 13.9.1 General

Iteration statements repeatedly execute an embedded statement.

iteration-statement: while-statement do-statement for-statement foreach-statement

#### 13.9.2 The while statement

The while statement conditionally executes an embedded statement zero or more times.

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

A while statement is executed as follows:

- The boolean-expression (§12.21) is evaluated.
- If the Boolean expression yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), control is transferred to the beginning of the while statement.
- If the Boolean expression yields false, control is transferred to the end point of the while statement.

Within the embedded statement of a while statement, a break statement (§13.10.2) may be used to transfer control to the end point of the while statement (thus ending iteration of the embedded statement), and a continue statement (§13.10.3) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the while statement).

The embedded statement of a while statement is reachable if the while statement is reachable and the Boolean expression does not have the constant value false.

The end point of a while statement is reachable if at least one of the following is true:

- The while statement contains a reachable break statement that exits the while statement.
- The while statement is reachable and the Boolean expression does not have the constant value true.

# 13.9.3 The do statement

The do statement conditionally executes an embedded statement one or more times.

```
do-statement:
    do embedded-statement while ( boolean-expression );
```

A do statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the *boolean-expression* (§12.21) is evaluated. If the Boolean expression yields true, control is transferred to the beginning of the do statement. Otherwise, control is transferred to the beginning of the do statement.

Within the embedded statement of a do statement, a break statement (§13.10.2) may be used to transfer control to the end point of the do statement (thus ending iteration of the embedded statement), and a continue statement (§13.10.3) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the do statement).

The embedded statement of a do statement is reachable if the do statement is reachable.

The end point of a do statement is reachable if at least one of the following is true:

- The do statement contains a reachable break statement that exits the do statement.
- The end point of the embedded statement is reachable and the Boolean expression does not have the constant value true.

#### 13.9.4 The for statement

The for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

```
for-statement:

for ( for-initializer<sub>opt</sub> ; for-condition<sub>opt</sub> ; for-iterator<sub>opt</sub> ) embedded-statement

for-initializer:

local-variable-declaration

statement-expression-list

for-condition:

boolean-expression

for-iterator:

statement-expression-list

statement-expression-list:

statement-expression

statement-expression

statement-expression-list , statement-expression
```

The *for-initializer*, if present, consists of either a *local-variable-declaration* (§13.6.2) or a list of *statement-expressions* (§13.7) separated by commas. The scope of a local variable declared by a *for-initializer* starts at the *local-variable-declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for-condition* and the *for-iterator*.

The for-condition, if present, shall be a boolean-expression (§12.21).

The for-iterator, if present, consists of a list of statement-expressions (§13.7) separated by commas.

A for statement is executed as follows:

- If a *for-initializer* is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- If a *for-condition* is present, it is evaluated.
- If the *for-condition* is not present or if the evaluation yields true, control is transferred to the
  embedded statement. When and if control reaches the end point of the embedded statement (possibly
  from execution of a continue statement), the expressions of the *for-iterator*, if any, are evaluated in
  sequence, and then another iteration is performed, starting with evaluation of the *for-condition* in the
  step above.
- If the *for-condition* is present and the evaluation yields false, control is transferred to the end point of the for statement.

Within the embedded statement of a for statement, a break statement (§13.10.2) 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 (§13.10.3) may be used to transfer control to the end point of the embedded statement (thus executing the *for-iterator* and performing another iteration of the for statement, starting with the *for-condition*).

The embedded statement of a for statement is reachable if one of the following is true:

- The for statement is reachable and no *for-condition* is present.
- The for statement is reachable and a *for-condition* is present and does not have the constant value false.

The end point of a for statement is reachable if at least one of the following is true:

- The for statement contains a reachable break statement that exits the for statement.
- The for statement is reachable and a *for-condition* is present and does not have the constant value true.

### 13.9.5 The foreach statement

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

## foreach-statement:

foreach ( local-variable-type identifier in expression ) embedded-statement

The *local-variable-type* and *identifier* of a foreach statement declare the *iteration variable* of the statement. If the var identifier is given as the *local-variable-type*, and no type named var is in scope, the iteration variable is said to be an *implicitly typed iteration variable*, and its type is taken to be the element type of the foreach statement, as specified below. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a foreach statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (via assignment or the ++ and -- operators) or pass the iteration variable as a ref or out parameter.

In the following, for brevity, IEnumerable, IEnumerator, IEnumerable<T> and IEnumerator<T> refer to the corresponding types in the namespaces System.Collections and System.Collections.Generic.

The compile-time processing of a foreach statement first determines the *collection type*, *enumerator type* and *element type* of the expression. This determination proceeds as follows:

- If the type X of *expression* is an array type then there is an implicit reference conversion from X to the IEnumerable interface (since System.Array implements this interface). The *collection type* is the IEnumerable interface, the *enumerator type* is the IEnumerator interface and the *element type* is the element type of the array type X.
- If the type X of *expression* is dynamic then there is an implicit conversion from *expression* to the IEnumerable interface (§11.2.9). The *collection type* is the IEnumerable interface and the *enumerator type* is the IEnumerator interface. If the var identifier is given as the *local-variable-type* then the *element type* is dynamic, otherwise it is object.
- Otherwise, determine whether the type X has an appropriate GetEnumerator method:
  - Perform member lookup on the type X with identifier GetEnumerator and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
  - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous public instance method or no applicable methods.
  - If the return type E of the GetEnumerator method is not a class, struct or interface type, an error is produced and no further steps are taken.
  - Member lookup is performed on E with the identifier Current and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.

- Member lookup is performed on E with the identifier MoveNext and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
- Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not bool, an error is produced and no further steps are taken.
- The *collection type* is X, the *enumerator type* is E, and the *element type* is the type of the Current property.
- Otherwise, check for an enumerable interface:
  - If among all the types T<sub>i</sub> for which there is an implicit conversion from X to IEnumerable<T<sub>i</sub>>, there is a unique type T such that T is not dynamic and for all the other T<sub>i</sub> there is an implicit conversion from IEnumerable<T> to IEnumerable<T<sub>i</sub>>, then the *collection type* is the interface IEnumerable<T>, the *enumerator type* is the interface IEnumerator<T>, and the *element type* is T.
  - Otherwise, if there is more than one such type T, then an error is produced and no further steps are taken.
  - Otherwise, if there is an implicit conversion from X to the System.Collections.IEnumerable interface, then the *collection type* is this interface, the *enumerator type* is the interface System.Collections.IEnumerator, and the *element type* is object.
  - $\circ$   $\;$  Otherwise, an error is produced and no further steps are taken.

The above steps, if successful, unambiguously produce a collection type C, enumerator type E and element type T. A foreach statement of the form

foreach (V v in x) embedded-statement

is then expanded to:

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded-statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

The variable e is not visible to or accessible to the expression x or the embedded statement or any other source code of the program. The variable v is read-only in the embedded statement. If there is not an explicit conversion (§11.2.13) from T (the element type) to V (the *local-variable-type* in the foreach statement), an error is produced and no further steps are taken. [*Note*: If x has the value null, a System.NullReferenceException is thrown at run-time. *end note*]

An implementation is permitted to implement a given *foreach-statement* differently; e.g., for performance reasons, as long as the behavior is consistent with the above expansion.

The placement of v inside the while loop is important for how it is captured (\$12.16.6.2) by any anonymous function occurring in the *embedded-statement*.

[Example:

int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
 if (f == null) f = () => Console.WriteLine("First value: " + value);
}
f();

If v in the expanded form were declared outside of the while loop, it would be shared among all iterations, and its value after the for loop would be the final value, 13, which is what the invocation of f would print. Instead, because each iteration has its own variable v, the one captured by f in the first iteration will continue to hold the value 7, which is what will be printed. (Note that earlier versions of C# declared v outside of the while loop.) *end example*]

The body of the finally block is constructed according to the following steps:

- If there is an implicit conversion from E to the System. IDisposable interface, then
  - If E is a non-nullable value type then the finally clause is expanded to the semantic equivalent of:
     finally {
     ((system IDisposable))

```
((System.IDisposable)e).Dispose();
}
```

o Otherwise the finally clause is expanded to the semantic equivalent of:

```
finally {
   System.IDisposable d = e as System.IDisposable;
   if (d != null) d.Dispose();
}
```

except that if E is a value type, or a type parameter instantiated to a value type, then the conversion of e to System.IDisposable shall not cause boxing to occur.

• Otherwise, if E is a sealed type, the finally clause is expanded to an empty block:

```
finally {
}
```

• Otherwise, the finally clause is expanded to:

```
finally {
   System.IDisposable d = e as System.IDisposable;
   if (d != null) d.Dispose();
}
```

The local variable d is not visible to or accessible to any user code. In particular, it does not conflict with any other variable whose scope includes the finally block.

The order in which foreach traverses the elements of an array, is as follows: For single-dimensional arrays elements are traversed in increasing index order, starting with index 0 and ending with index Length – 1. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

[Example: The following example prints out each value in a two-dimensional array, in element order:

```
using System;
class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };
        foreach (double elementValue in values)
            Console.write("{0} ", elementValue);
        Console.writeLine();
    }
}
```

The output produced is as follows:

1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9

end example]

[Example: In the following example

int[] numbers = { 1, 3, 5, 7, 9 }; foreach (var n in numbers) Console.WriteLine(n);

the type of n is inferred to be int, the element type of numbers.

end example]

#### 13.10 Jump statements

#### 13.10.1 General

Jump statements unconditionally transfer control.

jump-statement: break-statement continue-statement goto-statement return-statement throw-statement

The location to which a jump statement transfers control is called the *target* of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to *exit* the block. While a jump statement can transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening try statements. In the absence of such try statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening try statements, execution is more complex. If the jump statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.

[Example: In the following code

```
using System;
class Test
   static void Main() {
      while (true) {
         try {
            try
               Console.WriteLine("Before break");
               break;
            finally {
               Console.WriteLine("Innermost finally block");
            }
         finally {
            Console.WriteLine("Outermost finally block");
      Console.WriteLine("After break");
   }
}
```

the finally blocks associated with two try statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

Before break Innermost finally block Outermost finally block After break

end example]

#### 13.10.2 The break statement

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement.

```
break-statement:
    break ;
```

The target of a break statement is the end point of the nearest enclosing switch, while, do, for, or foreach statement. If a break statement is not enclosed by a switch, while, do, for, or foreach statement, a compile-time error occurs.

When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement (§13.10.4) shall be used.

A break statement cannot exit a finally block (§13.11). When a break statement occurs within a finally block, the target of the break statement shall be within the same finally block; otherwise a compile-time error occurs.

A break statement is executed as follows:

- If the break statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the break statement.

Because a break statement unconditionally transfers control elsewhere, the end point of a break statement is never reachable.

#### 13.10.3 The continue statement

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement.

continue-statement:
 continue ;

The target of a continue statement is the end point of the embedded statement of the nearest enclosing while, do, for, or foreach statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs.

When multiple while, do, for, or foreach statements are nested within each other, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement (§13.10.4) shall be used.

A continue statement cannot exit a finally block (§13.11). When a continue statement occurs within a finally block, the target of the continue statement shall be within the same finally block; otherwise a compile-time error occurs.

A continue statement is executed as follows:

- If the continue statement exits one or more try blocks with associated finally blocks, control is
  initially transferred to the finally block of the innermost try statement. When and if control
  reaches the end point of a finally block, control is transferred to the finally block of the next
  enclosing try statement. This process is repeated until the finally blocks of all intervening try
  statements have been executed.
- Control is transferred to the target of the continue statement.

Because a continue statement unconditionally transfers control elsewhere, the end point of a continue statement is never reachable.

#### 13.10.4 The goto statement

The goto statement transfers control to a statement that is marked by a label.

```
goto-statement:
  goto identifier ;
  goto case constant-expression ;
  goto default ;
```

The target of a goto *identifier* statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. [*Note*: This rule permits the use of a goto statement to transfer control *out of* a nested scope, but not *into* a nested scope. In the example

```
using System;
class Test
   static void Main(string[] args) {
       string[,] table = {
    {"Red", "Blue", "Green"},
    {"Monday", "Wednesday", "Friday"}
       };
       foreach (string str in args) {
           int row, colm;
           for (row = 0; row <= 1; ++row)
               for (colm = 0; colm <= 2; ++colm)
    if (str == table[row,colm])</pre>
                       goto done;
           Console.WriteLine("{0} not found", str);
           continue;
   done:
           Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
       }
   }
}
```

a goto statement is used to transfer control out of a nested scope. end note]

The target of a goto case statement is the statement list in the immediately enclosing switch statement (§13.8.3) 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 (§11.2) to the governing type of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant value, a compile-time error occurs.

The target of a goto default statement is the statement list in the immediately enclosing switch statement (§13.8.3), 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.

A goto statement cannot exit a finally block (§13.11). When a goto statement occurs within a finally block, the target of the goto statement shall be within the same finally block, or otherwise a compile-time error occurs.

A goto statement is executed as follows:

- If the goto statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the goto statement.

Because a goto statement unconditionally transfers control elsewhere, the end point of a goto statement is never reachable.

#### 13.10.5 The return statement

The return statement returns control to the current caller of the function member in which the return statement appears.

return-statement: return expression<sub>opt</sub> ;

A function member is said to *compute a value* if it is a method with a non-void result type (§15.6.11), the get accessor of a property or indexer, or a user-defined operator. Function members that do not compute a value are methods with the effective return type void, set accessors of properties and indexers, add and remove accessors of event, instance constructors, static constructors and finalizers.

Within a function member, a return statement with no expression can only be used if the function member does not compute a value. Within a function member, a return statement with an expression can only be used if the function member computes a value. Where the return statement includes an expression, an implicit conversion (§11.2) shall exist from the type of the expression to the effective return type of the containing function member.

Return statements can also be used in the body of anonymous function expressions (§12.16), and participate in determining which conversions exist for those functions (§11.7.1).

It is a compile-time error for a return statement to appear in a finally block (§13.11).

A return statement is executed as follows:

- If the return statement specifies an expression, the expression is evaluated and its value is converted to the effective return type of the containing function by an implicit conversion. The result of the conversion becomes the result value produced by the function.
- If the return statement is enclosed by one or more try or catch blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all enclosing try statements have been executed.
- If the containing function is not an async function, control is returned to the caller of the containing function along with the result value, if any.
- If the containing function is an async function, control is returned to the current caller, and the result value, if any, is recorded in the return task as described in (§15.15.2).

Because a return statement unconditionally transfers control elsewhere, the end point of a return statement is never reachable.

#### 13.10.6 The throw statement

The throw statement throws an exception.

throw-statement: throw expression<sub>opt</sub>;

A throw statement with an expression throws an exception produced by evaluating the expression. The expression shall be implicitly convertible to System.Exception, and the result of evaluating the expression is converted to System.Exception before being thrown. If the result of the conversion is null, a System.NullReferenceException is thrown instead.

A throw statement with no expression can be used only in a catch block, in which case, that statement re-throws the exception that is currently being handled by that catch block.

Because a throw statement unconditionally transfers control elsewhere, the end point of a throw statement is never reachable.

When an exception is thrown, control is transferred to the first catch clause in an enclosing try statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as **exception propagation**. Propagation of an exception consists of repeatedly evaluating the following steps until a catch clause that matches the exception is found. In this description, the **throw point** is initially the location at which the exception is thrown.

- In the current function member, each try statement that encloses the throw point is examined. For each statement S, starting with the innermost try statement and ending with the outermost try statement, the following steps are evaluated:
  - If the try block of S encloses the throw point and if S has one or more catch clauses, the catch clauses are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies an exception type T (or a type parameter that at run-time denotes an exception type T) such that the run-time type of E derives from T is considered a match. A general catch (§13.11) clause is considered a match for any exception type. If a matching catch clause is located, the exception propagation is completed by transferring control to the block of that catch clause.
  - Otherwise, if the try block or a catch block of S encloses the throw point and if S has a finally block, control is transferred to the finally block. If the finally block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the finally block, processing of the current exception is continued.
- If an exception handler was not located in the current function invocation, the function invocation is terminated, and one of the following occurs:
  - If the current function is non-async, the steps above are repeated for the caller of the function with a throw point corresponding to the statement from which the function member was invoked.
  - If the current function is async and task-returning, the exception is recorded in the return task, which is put into a faulted or cancelled state as described in §15.15.2.
  - If the current function is async and void-returning, the synchronization context of the current thread is notified as described in §15.15.3.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

## 13.11 The try statement

The try statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the try statement provides the ability to specify a block of code that is always executed when control leaves the try statement.

try-statement: try block catch-clauses try block catch-clauses specific-catch-clauses specific-catch-clauses specific-catch-clauses: specific-catch-clauses: specific-catch-clause specific-catch-clauses specific-catch-clause specific-catch-clauses specific-catch-clause specific-catch-clause: catch ( type identifier<sub>opt</sub> ) block general-catch-clause: catch block finally-clause: finally block

There are three possible forms of try statements:

- A try block followed by one or more catch blocks.
- A try block followed by a finally block.
- A try block followed by one or more catch blocks followed by a finally block.

When a catch clause specifies a *type*, the type shall be System.Exception or a type that derives from System.Exception. When a catch clause specifies a *type-parameter* it shall be a type parameter type whose effective base class is or derives from System.Exception.

When a catch clause specifies both a *class-type* and an *identifier*, an *exception variable* of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the catch block. During execution of the catch block, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a catch clause includes an exception variable name, it is impossible to access the exception object in the catch block.

A catch clause that specifies neither an exception type nor an exception variable name is called a general catch clause. A try statement can only have one general catch clause, and, if one is present, it shall be the last catch clause.

[Note: Some programming languages might support exceptions that are not representable as an object derived from System.Exception, although such exceptions could never be generated by C# code. A general catch clause might be used to catch such exceptions. Thus, a general catch clause is semantically different from one that specifies the type System.Exception, in that the former might also catch exceptions from other languages. end note]

In order to locate a handler for an exception, catch clauses are examined in lexical order. A compile-time error occurs if a catch clause specifies a type that is the same as, or is derived from, a type that was specified in an earlier catch clause for the same try. [*Note*: Without this restriction, it would be possible to write unreachable catch clauses. *end note*]

Within a catch block, a throw statement (§13.10.6) with no expression can be used to re-throw the exception that was caught by the catch block. Assignments to an exception variable do not alter the exception that is re-thrown.

[Example: In the following code

```
using System;
class Test
{
    static void F() {
        try
            ′{
G();
        }
       }
catch (Exception e) {
    Console.WriteLine("Exception in F: " + e.Message);
    e = new Exception("F");
    throw; // re-throw
    }
    static void G() {
        throw new Exception("G");
    }
    static void Main() {
        try {
F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

the method F catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output produced is:

Exception in F: G Exception in Main: G

If the first catch block had thrown e instead of rethrowing the current exception, the output produced would be as follows:

Exception in F: G Exception in Main: F

#### end example]

It is a compile-time error for a break, continue, or goto statement to transfer control out of a finally block. When a break, continue, or goto statement occurs in a finally block, the target of the statement shall be within the same finally block, or otherwise a compile-time error occurs.

It is a compile-time error for a return statement to occur in a finally block.

A try statement is executed as follows:

- Control is transferred to the try block.
- When and if control reaches the end point of the try block:
  - o If the try statement has a finally block, the finally block is executed.
  - Control is transferred to the end point of the try statement.
- If an exception is propagated to the try statement during execution of the try block:
  - The catch clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception

type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located:

- If the matching catch clause declares an exception variable, the exception object is assigned to the exception variable.
- Control is transferred to the matching catch block.
- When and if control reaches the end point of the catch block:
  - If the try statement has a finally block, the finally block is executed.
  - Control is transferred to the end point of the try statement.
- If an exception is propagated to the try statement during execution of the catch block:
  - If the try statement has a finally block, the finally block is executed.
  - The exception is propagated to the next enclosing try statement.
- o If the try statement has no catch clauses or if no catch clause matches the exception:
  - If the try statement has a finally block, the finally block is executed.
  - The exception is propagated to the next enclosing try statement.

The statements of a finally block are always executed when control leaves a try statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a break, continue, goto, or return statement, or as a result of propagating an exception out of the try statement.

If an exception is thrown during execution of a finally block, and is not caught within the same finally block, the exception is propagated to the next enclosing try statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the throw statement (§13.10.6).

The try block of a try statement is reachable if the try statement is reachable.

A catch block of a try statement is reachable if the try statement is reachable.

The finally block of a try statement is reachable if the try statement is reachable.

The end point of a try statement is reachable if both of the following are true:

- The end point of the try block is reachable or the end point of at least one catch block is reachable.
- If a finally block is present, the end point of the finally block is reachable.

#### 13.12 The checked and unchecked statements

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

checked-statement: checked block unchecked-statement: unchecked block

The checked statement causes all expressions in the *block* to be evaluated in a checked context, and the unchecked statement causes all expressions in the *block* to be evaluated in an unchecked context.

The checked and unchecked statements are precisely equivalent to the checked and unchecked operators (§12.7.14), except that they operate on blocks instead of expressions.

#### 13.13 The lock statement

The lock statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

lock-statement: lock (expression) embedded-statement

The expression of a lock statement shall denote a value of a type known to be a *reference*. No implicit boxing conversion (§11.2.8) is ever performed for the expression of a lock statement, and thus it is a compile-time error for the expression to denote a value of a *value-type*.

A lock statement of the form

lock (x) ...

where x is an expression of a *reference-type*, is precisely equivalent to:

```
bool __lockWasTaken = false;
try {
   System.Threading.Monitor.Enter(x, ref __lockWasTaken); ...
finally {
   if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

except that x is only evaluated once.

While a mutual-exclusion lock is held, code executing in the same execution thread can also obtain and release the lock. However, code executing in other threads is blocked from obtaining the lock until the lock is released.

#### 13.14 The using statement

The using statement obtains one or more resources, executes a statement, and then disposes of the resource.

```
using-statement:
using ( resource-acquisition ) embedded-statement
```

resource-acquisition: local-variable-declaration expression

A **resource** is a class or struct that implements the System.IDisposable interface, which includes a single parameterless method named Dispose. Code that is using a resource can call Dispose to indicate that the resource is no longer needed.

If the form of *resource-acquisition* is *local-variable-declaration* then the type of the *local-variable-declaration* shall be either dynamic or a type that can be implicitly converted to System.IDisposable. If the form of *resource-acquisition* is *expression* then this expression shall be implicitly convertible to System.IDisposable.

Local variables declared in a *resource-acquisition* are read-only, and shall include an initializer. A compiletime error occurs if the embedded statement attempts to modify these local variables (via assignment or the ++ and -- operators), take the address of them, or pass them as ref or out parameters.

A using statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a try statement that includes a finally clause. This finally clause disposes of the resource. If a null resource is acquired, then no call to Dispose is made, and no exception is thrown. If the resource is of type dynamic it is dynamically converted through an implicit dynamic conversion (§11.2.9) to IDisposable during acquisition in order to ensure that the conversion is successful before the usage and disposal.

A using statement of the form

using (ResourceType resource = expression) statement

corresponds to one of three possible expansions. When ResourceType is a non-nullable value type or a type parameter with the value type constraint (§15.2.5), the expansion is semantically equivalent to

```
{
  ResourceType resource = expression;
  try {
    statement;
  }
  finally {
    ((IDisposable)resource).Dispose();
  }
}
```

except that the cast of resource to System. IDisposable shall not cause boxing to occur.

Otherwise, when ResourceType is dynamic, the expansion is

```
{
    ResourceType resource = expression;
    IDisposable d = resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
Otherwise, the expansion is
    {
        ResourceType resource = expression;
        try {
    }
}
```

```
statement;
}
finally {
    IDisposable d = (IDisposable)resource;
    if (d != null) d.Dispose();
}
```

In any expansion, the resource variable is read-only in the embedded statement, and the d variable is inaccessible in, and invisible to, the embedded statement.

An implementation is permitted to implement a given using-statement differently, e.g., for performance reasons, as long as the behavior is consistent with the above expansion.

A using statement of the form:

using (expression) statement

has the same three possible expansions. In this case ResourceType is implicitly the compile-time type of the expression, if it has one. Otherwise the interface IDisposable itself is used as the ResourceType. The resource variable is inaccessible in, and invisible to, the embedded statement.

When a *resource-acquisition* takes the form of a *local-variable-declaration*, it is possible to acquire multiple resources of a given type. A using statement of the form

using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement

is precisely equivalent to a sequence of nested using statements:

using (ResourceType r1 = e1)
 using (ResourceType r2 = e2)
 ...
 using (ResourceType rN = eN)
 statement

[*Example*: The example below creates a file named log.txt and writes two lines of text to the file. The example then opens that same file for reading and copies the contained lines of text to the console.

```
using System;
using System.IO;
class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }
        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }
        }
        }
    }
}
```

Since the TextWriter and TextReader classes implement the IDisposable interface, the example can use using statements to ensure that the underlying file is properly closed following the write or read operations. *end example*]

#### 13.15 The yield statement

The yield statement is used in an iterator block (§13.3) to yield a value to the enumerator object (§15.14.5) or enumerable object (§15.14.6) of an iterator or to signal the end of the iteration.

yield-statement: yield return expression ; yield break ;

yield is a contextual keyword (§7.4.4) and has special meaning only when used immediately before a return or break keyword.

There are several restrictions on where a yield statement can appear, as described in the following.

- It is a compile-time error for a yield statement (of either form) to appear outside a *method-body*, *operator-body*, or *accessor-body*.
- It is a compile-time error for a yield statement (of either form) to appear inside an anonymous function.
- It is a compile-time error for a yield statement (of either form) to appear in the finally clause of a try statement.
- It is a compile-time error for a yield return statement to appear anywhere in a try statement that contains any *catch-clauses*.

[Example: The following example shows some valid and invalid uses of yield statements.

```
delegate IEnumerable<int> D();
```

```
IEnumerator<int> GetEnumerator() {
   try {
       yield return 1:
                                 // Ok
// Ok
       yield break;

}
finally {
    yield return 2;
    inald break;
}

                                 // Error, yield in finally
// Error, yield in finally
   }
   try {
    yield return 3;
                                 // Error, yield return in try/catch
       yield break;
                                 // Ok
   }
   catch {
       yield return 4;
                                 // Error, yield return in try/catch
       yield break;
                                 // Ok
   }
   D d = delegate {
       yield return 5;
                                // Error, yield in an anonymous function
   };
}
int MyMethod() {
   yield return 1;
                                 // Error, wrong return type for an
                                 // iterator block
}
```

```
end example]
```

An implicit conversion (§11.2) shall exist from the type of the expression in the yield return statement to the yield type (§15.14.4) of the iterator.

A yield return statement is executed as follows:

- The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the Current property of the enumerator object.
- Execution of the iterator block is suspended. If the yield return statement is within one or more try blocks, the associated finally blocks are *not* executed at this time.
- The MoveNext method of the enumerator object returns true to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object's MoveNext method resumes execution of the iterator block from where it was last suspended.

A yield break statement is executed as follows:

- If the yield break statement is enclosed by one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all enclosing try statements have been executed.
- Control is returned to the caller of the iterator block. This is either the MoveNext method or Dispose method of the enumerator object.

Because a yield break statement unconditionally transfers control elsewhere, the end point of a yield break statement is never reachable.

## 14. Namespaces

## 14.1 General

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Using directives (§14.5) are provided to facilitate the use of namespaces.

#### 14.2 Compilation units

A *compilation-unit* defines the overall structure of a source file. A compilation unit consists of zero or more *extern-alias-directives* followed by zero or more *using-directives* followed by zero or more *global-attributes* followed by zero or more *namespace-member-declarations*.

compilation-unit: extern-alias-directives<sub>opt</sub> using-directives<sub>opt</sub> global-attributes<sub>opt</sub> namespace-member-declarations<sub>opt</sub>

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *extern-alias-directives* of a compilation unit affect the *using-directives*, *global-attributes* and *namespace-member-declarations* of that compilation unit, but have no effect on other compilation units.

The *using-directives* of a compilation unit affect the *global-attributes* and *namespace-member-declarations* of that compilation unit, but have no effect on other compilation units.

The *global-attributes* (§22.3) of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The *namespace-member-declarations* of each compilation unit of a program contribute members to a single declaration space called the global namespace. [*Example*:

```
File A.cs:
    class A {}
File B.cs:
    class B {}
```

The two compilation units contribute to the single global namespace, in this case declaring two classes with the fully qualified names A and B. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name. *end example*]

## 14.3 Namespace declarations

A *namespace-declaration* consists of the keyword namespace, followed by a namespace name and body, optionally followed by a semicolon.

```
namespace-declaration:
namespace qualified-identifier namespace-body ; opt
```

qualified-identifier: identifier qualified-identifier . identifier

namespace-body:

{ extern-alias-directives<sub>opt</sub> using-directives<sub>opt</sub> namespace-member-declarations<sub>opt</sub> }

A namespace-declaration may occur as a top-level declaration in a *compilation-unit* or as a member declaration within another *namespace-declaration*. When a *namespace-declaration* occurs as a top-level declaration in a *compilation-unit*, the namespace becomes a member of the global namespace. When a *namespace-declaration* occurs within another *namespace-declaration*, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace shall be unique within the containing namespace.

Namespaces are implicitly public and the declaration of a namespace cannot include any access modifiers.

Within a *namespace-body*, the optional *using-directives* import the names of other namespaces and types, allowing them to be referenced directly instead of through qualified names. The optional *namespace-member-declarations* contribute members to the declaration space of the namespace.

The *qualified-identifier* of a *namespace-declaration* may be a single identifier or a sequence of identifiers separated by "." tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. [*Example*:

```
namespace N1.N2
{
class A {}
class B {}
}
```

is semantically equivalent to

```
namespace N1
{
namespace N2
{
class A {}
class B {}
}
```

#### end example]

Namespaces are open-ended, and two namespace declarations with the same fully qualified name (§8.8.2) contribute to the same declaration space (§8.3). [*Example*: In the following code

```
namespace N1.N2
{
    class A {}
}
namespace N1.N2
{
    class B {}
}
```

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names N1.N2.A and N1.N2.B. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name. *end example*]

## 14.4 Extern alias directives

An *extern-alias-directive* introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and applies also to nested namespaces of the aliased namespace.

extern-alias-directives: extern-alias-directive extern-alias-directives extern-alias-directive extern-alias-directive: extern alias identifier ;

The scope of an *extern-alias-directive* extends over the *using-directives*, *global-attributes* and *namespace-member-declarations* of its immediately containing *compilation-unit* or *namespace-body*.

Within a compilation unit or namespace body that contains an *extern-alias-directive*, the identifier introduced by the *extern-alias-directive* can be used to reference the aliased namespace. It is a compile-time error for the *identifier* to be the word global.

Within C# source code, a type is declared a member of a single namespace. However, a namespace hierarchy referenced by an extern alias may contain types that are also members of other namespaces. For example, if A and B are extern aliases, the names A::X, B::C.Y and global::D.Z may, depending on the external specification supported by the particular compiler, all refer to the same type.

The alias introduced by an *extern-alias-directive* is very similar to the alias introduced by a *using-alias-directive*. See §14.5.2 for more detailed discussion of *extern-alias-directives* and *using-alias-directives*.

alias is a contextual keyword (§7.4.4) and only has special meaning when it immediately follows the extern keyword in an *extern-alias-directive*. [*Example*: In fact an extern alias could use the identifier alias as its name:

extern alias alias;

end example]

#### 14.5 Using directives

#### 14.5.1 General

**Using directives** facilitate the use of namespaces and types defined in other namespaces. Using directives impact the name resolution process of *namespace-or-type-names* (§8.8) and *simple-names* (§12.7.3), but unlike declarations, *using-directives* do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

using-directives: using-directive using-directives using-directive

using-directive: using-alias-directive using-namespace-directive

A using-alias-directive (§14.5.2) introduces an alias for a namespace or type.

A using-namespace-directive (§14.5.3) imports the type members of a namespace.

The scope of a *using-directive* extends over the *namespace-member-declarations* of its immediately containing compilation unit or namespace body. The scope of a *using-directive* specifically does not include its peer *using-directives*. Thus, peer *using-directives* do not affect each other, and the order in which they are written is insignificant. In contrast, the scope of an *extern-alias-directive* includes the *using-directives* defined in the same compilation unit or namespace body.

#### 14.5.2 Using alias directives

A *using-alias-directive* introduces an identifier that serves as an alias for a namespace or type within the immediately enclosing compilation unit or namespace body.

```
using-alias-directive:
    using identifier = namespace-or-type-name ;
```

Within global attributes and member declarations in a compilation unit or namespace body that contains a *using-alias-directive*, the identifier introduced by the *using-alias-directive* can be used to reference the given namespace or type. [*Example*:

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using A = N1.N2.A;
    class B: A {}
}
```

Above, within member declarations in the N3 namespace, A is an alias for N1.N2.A, and thus class N3.B derives from class N1.N2.A. The same effect can be obtained by creating an alias R for N1.N2 and then referencing R.A:

```
namespace N3
{
    using R = N1.N2;
    class B: R.A {}
}
```

#### end example]

Within using directives, global attributes and member declarations in a compilation unit or namespace body that contains an *extern-alias-directive*, the identifier introduced by the *extern-alias-directive* can be used to reference the associated namespace. [*Example*: For example:

```
namespace N1
{
    extern alias N2;
    class B: N2::A {}
}
```

Above, within member declarations in the N1 namespace, N2 is an alias for some namespace whose definition is external to the source code of the program. Class N1.B derives from class N2.A. The same effect can be obtained by creating an alias A for N2.A and then referencing A:

```
namespace N1
{
    extern alias N2;
    using A = N2::A;
    class B: A {}
}
```

#### end example]

An *extern-alias-directive* or *using-alias-directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an alias directive is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs. [*Example*: In the following code

```
namespace N3
{
    extern alias R1;
    using R2 = N1.N2;
}
namespace N3
{
    class B: R1::A, R2.I {} // Error, R1 and R2 unknown
}
```

the scopes of the alias directives that introduce R1 and R2 only extend to member declarations in the namespace body in which they are contained, so R1 and R2 are unknown in the second namespace declaration. However, placing the alias directives in the containing compilation unit causes the alias to become available within both namespace declarations:

```
extern alias R1;
using R2 = N1.N2;
namespace N3
{
    class B: R1::A, R2.I {}
}
namespace N3
{
    class C: R1::A, R2.I {}
}
```

#### end example]

Each *extern-alias-directive* or *using-alias-directive* in a *compilation-unit* or *namespace-body* contributes a name to the *alias declaration space* (§8.3) of the immediately enclosing *compilation-unit* or *namespace-body*. The *identifier* of the alias directive shall be unique within the corresponding alias declaration space. The alias identifier need not be unique within the global declaration space or the declaration space of the corresponding namespace. [*Example*:

```
extern alias A;
extern alias B;
using A = N1.N2; // Error: alias A already exists
class B {} // Ok
```

The using alias named A causes an error since there is already an alias named A in the same compilation unit. The class named B does not conflict with the extern alias named B since these names are added to distinct declaration spaces. The former is added to the global declaration space and the latter is added to the alias declaration space for this compilation unit.

When an alias name matches the name of a member of a namespace, usage of either must be appropriately qualified:

```
namespace N1.N2
{
    class B {}
}
namespace N3
{
    class A {}
    class B : A {}
}
namespace N3
{
    using A = N1.N2;
    using B = N1.N2.B;
```

}

In the second namespace body for N3, unqualified use of B results in an error, since N3 contains a member named B and the namespace body that also declares an alias with name B; likewise for A. The class N3.B can be referenced as N3.B or global::N3.B. The alias A can be used in a *qualified-alias-member* (§14.8), such as A::B. The alias B is essentially useless. It cannot be used in a *qualified-alias-member* since only namespace aliases can be used in a *qualified-alias-member* and B aliases a type. *end example*]

Just like regular members, names introduced by *alias-directives* are hidden by similarly named members in nested scopes. [*Example*: In the following code

```
using R = N1.N2;
namespace N3
{
    class R {}
    class B: R.A {} // Error, R has no member A
}
```

the reference to R.A in the declaration of B causes a compile-time error because R refers to N3.R, not N1.N2. *end example*]

The order in which *extern-alias-directives* are written has no significance. Likewise, the order in which *using-alias-directives* are written has no significance, but all *using-alias-directives* must come after all *extern-alias-directives* in the same compilation unit or namespace body. Resolution of the *namespace-or-type-name* referenced by a *using-alias-directive* is not affected by the *using-alias-directive* itself or by other *using-directives* in the immediately containing compilation unit or namespace body, but may be affected by *extern-alias-directives* in the immediately containing compilation unit or namespace body. In other words, the *namespace-or-type-name* of a *using-alias-directive* is resolved as if the immediately containing compilation unit or namespace body. In other words, the *namespace-or-type-name* of a *using-alias-directive* is resolved as if the immediately containing *directives*. [*Example*: In the following code

```
namespace N1.N2 {}
namespace N3
{
    extern alias E;
    using R1 = E::N; // OK
    using R2 = N1; // OK
    using R3 = N1.N2; // OK
    using R4 = R2.N2; // Error, R2 unknown
}
```

the last *using-alias-directive* results in a compile-time error because it is not affected by the previous *using-alias-directive*. The first *using-alias-directive* does not result in an error since the scope of the extern alias E includes the *using-alias-directive*. *end example*]

A *using-alias-directive* can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type nested within that namespace.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name. [*Example*: Given

```
namespace N1.N2
{
    class A {}
}
```

```
namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;
    class B
    {
        N1.N2.A a; // refers to N1.N2.A
        R1.N2.A b; // refers to N1.N2.A
        R2.A c; // refers to N1.N2.A
    }
}
```

the names N1.N2.A, R1.N2.A, and R2.A are equivalent and all refer to the class declaration whose fully qualified name is N1.N2.A. *end example*]

Although each part of a partial type (§15.2.7) is declared within the same namespace, the parts are typically written within different namespace declarations. Thus, different extern alias directives and using directives can be present for each part. When interpreting simple names (§12.7.3) within one part, only the extern alias directives and using directives of the namespace bodies and compilation unit enclosing that part are considered. This may result in the same identifier having different meanings in different parts. [*Example*:

```
namespace N
   using List = System.Collections.ArrayList;
   partial class A
      List x;
                        // x has type System.Collections.ArrayList
   }
}
namespace N
ł
   using List = Widgets.LinkedList;
   partial class A
   ł
                        // y has type Widgets.LinkedList
      List y;
   }
}
```

```
end example]
```

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments. [*Example*:

```
namespace N1
ſ
   class A<T>
   ł
      class B {}
   3
}
namespace N2
Ł
                            // Error, cannot name unbound generic type
   using W = N1.A;
   using X = N1.A.B;
                            // Error, cannot name unbound generic type
                            // Ok, can name closed constructed type
   using Y = N1.A < int >;
   using Z < T > = N1.A < T >;
                            // Error, using alias cannot have type
parameters
```

```
end example]
```

#### 14.5.3 Using namespace directives

A *using-namespace-directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

using-namespace-directive: using namespace-name ;

Within member declarations in a compilation unit or namespace body that contains a *using-namespace-directive*, the types contained in the given namespace can be referenced directly. [*Example*:

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1.N2;
    class B: A {}
}
```

Above, within member declarations in the N3 namespace, the type members of N1.N2 are directly available, and thus class N3.B derives from class N1.N2.A. *end example*]

A *using-namespace-directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. [*Example*: In the following code

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1;
    class B: N2.A {} // Error, N2 unknown
}
```

the *using-namespace-directive* imports the types contained in N1, but not the namespaces nested in N1. Thus, the reference to N2. A in the declaration of B results in a compile-time error because no members named N2 are in scope. *end example*]

Unlike a *using-alias-directive*, a *using-namespace-directive* may import types whose identifiers are already defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using-namespace-directive* are hidden by similarly named members in the enclosing compilation unit or namespace body. [*Example*:

```
namespace N1.N2
{
    class A {}
    class B {}
}
namespace N3
{
    using N1.N2;
    class A {}
}
```

Here, within member declarations in the N3 namespace, A refers to N3.A rather than N1.N2.A. *end example*]

Because names may be ambiguous when more than one imported namespace introduces the same type name, a *using-alias-directive* is useful to disambiguate the reference. [*Example*: In the following code

```
namespace N1
{
    class A {}
}
namespace N2
{
    class A {}
}
namespace N3
{
    using N1;
    using N2;
    class B: A {}
}
// Error, A is ambiguous
}
```

both N1 and N2 contain a member A, and because N3 imports both, referencing A in N3 is a compile-time error. In this situation, the conflict can be resolved either through qualification of references to A, or by introducing a *using-alias-directive* that picks a particular A. For example:

```
namespace N3
{
    using N1;
    using N2;
    using A = N1.A;
    class B: A {} // A means N1.A
}
```

end example]

Like a *using-alias-directive*, a *using-namespace-directive* does not contribute any new members to the underlying declaration space of the compilation unit or namespace, but, rather, affects only the compilation unit or namespace body in which it appears.

The *namespace-name* referenced by a *using-namespace-directive* is resolved in the same way as the *namespace-or-type-name* referenced by a *using-alias-directive*. Thus, *using-namespace-directives* in the same compilation unit or namespace body do not affect each other and can be written in any order.

## 14.6 Namespace member declarations

type-declaration

A namespace-member-declaration is either a namespace-declaration (§14.3) or a type-declaration (§14.7).

```
namespace-member-declarations:

namespace-member-declaration

namespace-member-declarations namespace-member-declaration

namespace-member-declaration:

namespace-declaration
```

A compilation unit or a namespace body can contain *namespace-member-declarations*, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

## 14.7 Type declarations

A type-declaration is a class-declaration (§15.2), a struct-declaration (§16.2), an interface-declaration (§18.2), an enum-declaration (§19.2), or a delegate-declaration (§20.2).

type-declaration: class-declaration struct-declaration interface-declaration enum-declaration delegate-declaration

A *type-declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

When a type declaration for a type T occurs as a top-level declaration in a compilation unit, the fully qualified name (§8.8.2) of the type declaration is the same as the unqualified name of the declaration (§8.8.2). When a type declaration for a type T occurs within a namespace, class, or struct declaration, the fully qualified name (§8.8.3) of the type declarationis S.N, where S is the fully qualified name of the containing namespace, class, or struct declaration.

A type declared within a class or struct is called a nested type (§15.3.9).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (§8.5.2):

- Types declared in compilation units or namespaces can have public or internal access. The default is internal access.
- Types declared in classes can have public, protected internal, protected, internal, or private access. The default is private access.
- Types declared in structs can have public, internal, or private access. The default is private access.

## 14.8 Qualified alias member

#### 14.8.1 General

The *namespace alias qualifier* : : makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The namespace alias qualifier always appears between two identifiers referred to as the left-hand and right-hand identifiers. Unlike the regular . qualifier, the left-hand identifier of the : : qualifier is looked up only as an extern or using alias.

A *qualified-alias-member* provides explicit access to the global namespace and to extern or using aliases that are potentially hidden by other entities.

qualified-alias-member: identifier :: identifier type-argument-list<sub>opt</sub>

A qualified-alias-member can be used as a namespace-or-type-name (§8.8) or as the left operand in a member-access (§12.7.5).

A *qualified-alias-member* consists of two identifiers, referred to as the left-hand and right-hand identifiers, seperated by the :: token and optionally followed by a *type-argument-list*. When the left-hand identifier is global then the global namespace is searched for the right-hand identifier. For any other left-hand identifier, that identifier is looked up as an extern or using alias (§14.4 and §14.5.2). A compile-time error occurs if there is no such alias or the alias references a type. If the alias references a namespace then that namespace is searched for the right-hand identifier.

A qualified-alias-member has one of two forms:

- N::I<A1, ..., AK>, where N and I represent identifiers, and <A1, ..., AK> is a type argument list. (K is always at least one.)
- N::I, where N and I represent identifiers. (In this case, K is considered to be zero.)

Using this notation, the meaning of a *qualified-alias-member* is determined as follows:

- If N is the identifier global, then the global namespace is searched for I:
  - If the global namespace contains a namespace named I and K is zero, then the *qualified-aliasmember* refers to that namespace.
  - Otherwise, if the global namespace contains a non-generic type named I and K is zero, then the *qualified-alias-member* refers to that type.
  - Otherwise, if the global namespace contains a type named I that has K type parameters, then the *qualified-alias-member* refers to that type constructed with the given type arguments.
  - Otherwise, the *qualified-alias-member* is undefined and a compile-time error occurs.
- Otherwise, starting with the namespace declaration (§14.3) immediately containing the *qualified-alias-member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified-alias-member*, the following steps are evaluated until an entity is located:
  - If the namespace declaration or compilation unit contains a *using-alias-directive* that associates N with a type, then the *qualified-alias-member* is undefined and a compile-time error occurs.
  - Otherwise, if the namespace declaration or compilation unit contains an *extern-alias-directive* or *using-alias-directive* that associates N with a namespace, then:
    - If the namespace associated with N contains a namespace named I and K is zero, then the *qualified-alias-member* refers to that namespace.
    - Otherwise, if the namespace associated with N contains a non-generic type named I and K is zero, then the *qualified-alias-member* refers to that type.
    - Otherwise, if the namespace associated with N contains a type named I that has K type parameters, then the *qualified-alias-member* refers to that type constructed with the given type arguments.
    - Otherwise, the *qualified-alias-member* is undefined and a compile-time error occurs.
- Otherwise, the *qualified-alias-member* is undefined and a compile-time error occurs.

[*Example*: In the code:

```
using S = System.Net.Sockets;
class A {
    public static int x;
}
class C {
    public void F(int A, object S) {
        // Use global::A.x instead of A.x
        global::A.x += A;
        // Use S::Socket instead of S.Socket
        S::Socket s = S as S::Socket;
    }
}
```

the class A is referenced with global:: A and the type System.Net.Sockets.Socket is referenced with S::Socket.Using A.x and S.Socket instead would have caused compile-time errors because A and S would have resolved to the parameters. *end example*]

[*Note*: The identifier global has special meaning only when used as the left-hand identifier of a *qualified-alias-name*. It is not a keyword and it is not itself an alias; it is a contextual keyword (§7.4.4). In the code:

```
class A { }
class C {
   global.A x; // Error: global is not defined
   global::A y; // Valid: References A in the global namespace
}
```

using global. A causes a compile-time error since there is no entity named global in scope. If some entity named global were in scope, then global in global. A would have resolved to that entity.

Using global as the left-hand identifier of a *qualified-alias-member* always causes a lookup in the global namespace, even if there is a using alias named global. In the code:

```
using global = MyGlobalTypes;
class A { }
class C {
  global.A x; // Valid: References MyGlobalTypes.A
  global::A y; // Valid: References A in the global namespace
}
```

global. A resolves to MyGlobalTypes. A and global: : A resolves to class A in the global namespace. *end note*]

#### 14.8.2 Uniqueness of aliases

Each compilation unit and namespace body has a separate declaration space for extern aliases and using aliases. Thus, while the name of an extern alias or using alias shall be unique within the set of extern aliases and using aliases declared in the immediately containing compilation unit or namespace body, an alias is permitted to have the same name as a type or namespace as long as it is used only with the :: qualifier.

[*Example*: In the following:

```
namespace N
{
   public class A {}
   public class B {}
}
namespace N
{
   using A = System.IO;
   class X
   {
      A.Stream s1; // Error, A is ambiguous
      A::Stream s2; // Ok
   }
}
```

the name A has two possible meanings in the second namespace body because both the class A and the using alias A are in scope. For this reason, use of A in the qualified name A.Stream is ambiguous and causes a compile-time error to occur. However, use of A with the :: qualifier is not an error because A is looked up only as a namespace alias. *end example*]

# 15. Classes

## 15.1 General

A class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, finalizers, and static constructors), and nested types. Class types support inheritance, a mechanism whereby a *derived class* can extend and specialize a *base class*.

## 15.2 Class declarations

#### 15.2.1 General

A class-declaration is a type-declaration (§14.7) that declares a new class.

class-declaration: attributes<sub>opt</sub> class-modifiers<sub>opt</sub> partial<sub>opt</sub> class identifier type-parameter-list<sub>opt</sub> class-base<sub>opt</sub> type-parameter-constraints-clauses<sub>opt</sub> class-body ;<sub>opt</sub>

A *class-declaration* consists of an optional set of *attributes* (§22), followed by an optional set of *class-modifiers* (§15.2.2), followed by an optional partial modifier (§15.2.7), followed by the keyword class and an *identifier* that names the class, followed by an optional *type-parameter-list* (§15.2.3), followed by an optional *class-base* specification (§15.2.4), followed by an optional set of *type-parameter-constraints-clauses* (§15.2.5), followed by a *class-body* (§15.2.6), optionally followed by a semicolon.

A class declaration shall not supply a *type-parameter-constraints-clauses* unless it also supplies a *type-parameter-list*.

A class declaration that supplies a *type-parameter-list* is a generic class declaration. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration, since type arguments for the containing type shall be supplied to create a constructed type.

## 15.2.2 Class modifiers

#### 15.2.2.1 General

A class-declaration may optionally include a sequence of class modifiers:

```
class-modifiers:
    class-modifier
    class-modifiers class-modifier
class-modifier:
    new
    public
    protected
    internal
    private
    abstract
    sealed
    static
```

It is a compile-time error for the same modifier to appear multiple times in a class declaration.

The new modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name, as described in §15.3.5. It is a compile-time error for the new modifier to appear on a class declaration that is not a nested class declaration.

The public, protected, internal, and private modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers might not be permitted (§8.5.2).

When a partial type declaration (§15.2.7) includes an accessibility specification (via the public, protected, internal, and private modifiers), that specification shall agree with all other parts that include an accessibility specification. If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility (§8.5.2).

The abstract, sealed, and static modifiers are discussed in the following subclauses.

#### 15.2.2.2 Abstract classes

The abstract modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An **abstract class** differs from a **non-abstract class** in the following ways:

- An abstract class cannot be instantiated directly, and it is a compile-time error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class shall include actual implementations of all inherited abstract members, thereby overriding those abstract members. [*Example*: In the following code

```
abstract class A
{
    public abstract void F();
}
abstract class B: A
{
    public void G() {}
}
class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

the abstract class A introduces an abstract method F. Class B introduces an additional method G, but since it doesn't provide an implementation of F, B shall also be declared abstract. Class C overrides F and provides an actual implementation. Since there are no abstract members in C, C is permitted (but not required) to be non-abstract. *end example*]

If one or more parts of a partial type declaration (§15.2.7) of a class include the abstract modifier, the class is abstract. Otherwise, the class is non-abstract.

#### 15.2.2.3 Sealed classes

The sealed modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

[*Note*: The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations. *end note*]

If one or more parts of a partial type declaration (§15.2.7) of a class include the sealed modifier, the class is sealed. Otherwise, the class is unsealed.

#### 15.2.2.4 Static classes

#### 15.2.2.4.1 General

The static modifier is used to mark the class being declared as a *static class*. A static class shall not be instantiated, shall not be used as a type and shall contain only static members. Only a static class can contain declarations of extension methods (§15.6.10).

A static class declaration is subject to the following restrictions:

- A static class shall not include a sealed or abstract modifier. (However, since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.)
- A static class shall not include a *class-base* specification (§15.2.4) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type object.
- A static class shall only contain static members (§15.3.8). [*Note*: All constants and nested types are classified as static members. *end note*]
- A static class shall not have members with protected or protected internal declared accessibility.

It is a compile-time error to violate any of these restrictions.

A static class has no instance constructors. It is not possible to declare an instance constructor in a static class, and no default instance constructor (§15.11.5) is provided for a static class.

The members of a static class are not automatically static, and the member declarations shall explicitly include a static modifier (except for constants and nested types). When a class is nested within a static outer class, the nested class is not a static class unless it explicitly includes a static modifier.

If one or more parts of a partial type declaration (§15.2.7) of a class include the static modifier, the class is static. Otherwise, the class is not static.

#### 15.2.2.4.2 Referencing static class types

A namespace-or-type-name (§8.8) is permitted to reference a static class if

- The *namespace-or-type-name* is the T in a *namespace-or-type-name* of the form T.I, or
- The namespace-or-type-name is the T in a typeof-expression (§12.7.12) of the form typeof(T).

A primary-expression (§12.7) is permitted to reference a static class if

• The primary-expression is the E in a member-access (§12.7.5) of the form E.I.

In any other context, it is a compile-time error to reference a static class. [*Note*: For example, it is an error for a static class to be used as a base class, a constituent type (§15.3.7) of a member, a generic type argument, or a type parameter constraint. Likewise, a static class cannot be used in an array type, a pointer type, a new expression, a cast expression, an is expression, an as expression, a sizeof expression, or a default value expression. *end note*]

## 15.2.3 Type parameters

A type parameter is a simple identifier that denotes a placeholder for a type argument supplied to create a constructed type. By constrast, a type argument (§9.4.2) is the type that is substituted for the type parameter when a constructed type is created.

type-parameter-list: < type-parameters > type-parameters: attributes<sub>opt</sub> type-parameter type-parameters , attributes<sub>opt</sub> type-parameter

type-parameter is defined in §9.5.

Each type parameter in a class declaration defines a name in the declaration space (§8.3) of that class. Thus, it cannot have the same name as another type parameter of that class or a member declared in that class. A type parameter cannot have the same name as the type itself.

Two partial generic type declarations (in the same program) contribute to the same unbound generic type if they have the same fully qualified name (which includes a *generic-dimension-specifier* (§12.7.12) for the number of type parameters) (§8.8.3). Two such partial type declarations shall specify the same name for each type parameter, in order.

#### 15.2.4 Class base specification

#### 15.2.4.1 General

A class declaration may include a *class-base* specification, which defines the direct base class of the class and the interfaces (§18) directly implemented by the class.

class-base:

: class-type : interface-type-list : class-type , interface-type-list

interface-type-list: interface-type interface-type-list , interface-type

#### 15.2.4.2 Base classes

When a *class-type* is included in the *class-base*, it specifies the direct base class of the class being declared. If a non-partial class declaration has no *class-base*, or if the *class-base* lists only interface types, the direct base class is assumed to be object. When a partial class declaration includes a base class specification, that base class specification shall reference the same type as all other parts of that partial type that include a base class specification. If no part of a partial class includes a base class specification, the base class is object. A class inherits members from its direct base class, as described in §15.3.4.

[Example: In the following code

class A {} class B: A {}

class A is said to be the direct base class of B, and B is said to be derived from A. Since A does not explicitly specify a direct base class, its direct base class is implicitly object. *end example*]

For a constructed class type, including a nested type declared within a generic type declaration (§16.3.9.7), if a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type-parameter* in the base class declaration, the corresponding *type-argument* of the constructed type. [*Example*: Given the generic class declarations

```
class B<U,V> {...}
class G<T>: B<string,T[]> {...}
```

the base class of the constructed type G<int> would be B<string, int[]>. end example]

The base class specified in a class declaration can be a constructed class type (§9.4). A base class cannot be a type parameter on its own (§9.5), though it can involve the type parameters that are in scope. [*Example*:

```
class Base<T> {}
```

end example]

The direct base class of a class type shall be at least as accessible as the class type itself (§8.5.5). For example, it is a compile-time error for a public class to derive from a private or internal class.

The direct base class of a class type shall not be any of the following types: System.Array, System.Delegate, System.Enum, or System.ValueType. Furthermore, a generic class declaration shall not use System.Attribute as a direct or indirect base class (§22.2.1).

In determining the meaning of the direct base class specification A of a class B, the direct base class of B is temporarily assumed to be object, which ensures that the meaning of a base class specification cannot recursively depend on itself. [*Example*: The following

```
class X<T> {
    public class Y{}
}
class Z : X<Z.Y> {}
```

Is in error since in the base class specification  $X < Z \cdot Y >$  the direct base class of Z is considered to be object, and hence (by the rules of §8.8) Z is not considered to have a member Y. *end example*]

The base classes of a class are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. [*Example*: In the following:

```
class A {...}
class B<T>: A {...}
class C<T>: B<IComparable<T>> {...}
class D<T>: C<T[]> {...}
```

the base classes of D<int> are C<int[]>, B<IComparable<int[]>>, A, and object.

#### end example]

Except for class object, every class has exactly one direct base class. The object class has no direct base class and is the ultimate base class of all other classes.

It is a compile-time error for a class to depend on itself. For the purpose of this rule, a class **directly depends on** its direct base class (if any) and *directly depends on* the nearest enclosing class within which it is nested (if any). Given this definition, the complete set of classes upon which a class depends is the transitive closure of the *directly depends on* relationship.

[Example: The example

class A: A {}

Is erroneous because the class depends on itself. Likewise, the example

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Finally, the example

class A: B.C {}

```
class B: A
{
    public class C {}
}
```

results in a compile-time error because A depends on B.C (its direct base class), which depends on B (its immediately enclosing class), which circularly depends on A. *end example*]

A class does not depend on the classes that are nested within it. [Example: In the following code

```
class A
{
class B: A {}
}
```

B depends on A (because A is both its direct base class and its immediately enclosing class), but A does not depend on B (since B is neither a base class nor an enclosing class of A). Thus, the example is valid. *end example*]

It is not possible to derive from a sealed class. [Example: In the following code

```
sealed class A {}
class B: A {} // Error, cannot derive from a sealed class
```

class B is in error because it attempts to derive from the sealed class A. end example]

#### 15.2.4.3 Interface implementations

A *class-base* specification may include a list of interface types, in which case the class is said to implement the given interface types. For a constructed class type, including a nested type declared within a generic type declaration (§15.3.9.7), each implemented interface type is obtained by substituting, for each *type-parameter* in the given interface, the corresponding *type-argument* of the constructed type.

The set of interfaces for a type declared in multiple parts (§15.2.7) is the union of the interfaces specified on each part. A particular interface can only be named once on each part, but multiple parts can name the same base interface(s). There shall only be one implementation of each member of any given interface. [*Example*: In the following:

```
partial class C: IA, IB {...}
partial class C: IC {...}
partial class C: IA, IB {...}
```

the set of base interfaces for class C is IA, IB, and IC. end example]

Typically, each part provides an implementation of the interface(s) declared on that part; however, this is not a requirement. A part can provide the implementation for an interface declared on a different part. [*Example*:

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}
partial class X: IComparable
{
    ...
}
```

```
end example]
```

The base interfaces specified in a class declaration can be constructed interface types (§9.4, §18.2). A base interface cannot be a type parameter on its own, though it can involve the type parameters that are in scope. [*Example*: The following code illustrates how a class can implement and extend constructed types:

```
class C<U, V> {}
interface I1<V> {}
```

```
class D: C<string, int>, I1<string> {}
class E<T>: C<int, T>, I1<T> {}
```

end example]

Interface implementations are discussed further in §18.6.

#### 15.2.5 Type parameter constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type*-parameter-constraints-clauses.

type-parameter-constraints-clauses: type-parameter-constraints-clause type-parameter-constraints-clauses type-parameter-constraints-clause
type-parameter-constraints-clause: where type-parameter : type-parameter-constraints
type-parameter-constraints: primary-constraint secondary-constraints constructor-constraint primary-constraint , secondary-constraints primary-constraint , constructor-constraint secondary-constraints , constructor-constraint primary-constraint , secondary-constraints , constructor-constraint
primary-constraint: class-type class struct
<pre>secondary-constraints: interface-type type-parameter secondary-constraints , interface-type secondary-constraints , type-parameter constructor-constraint: new ( )</pre>

Each *type-parameter-constraints-clause* consists of the token where, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be at most one where clause for each type parameter, and the where clauses can be listed in any order. Like the get and set tokens in a property accessor, the where token is not a keyword.

The list of constraints given in a where clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, new().

A primary constraint can be a class type or the *reference type constraint* class or the *value type constraint* struct. A secondary constraint can be a *type-parameter* or *interface-type*.

The reference type constraint specifies that a type argument used for the type parameter shall be a reference type. All class types, interface types, delegate types, array types, and type parameters known to be a reference type (as defined below) satisfy this constraint.

The value type constraint specifies that a type argument used for the type parameter shall be a nonnullable value type. All non-nullable struct types, enum types, and type parameters having the value type constraint satisfy this constraint. Note that although classified as a value type, a nullable value type (§9.3.11) does not satisfy the value type constraint. A type parameter having the value type constraint shall not also have the *constructor-constraint*, although it may be used as a type argument for another type parameter with a *constructor-constraint*. [*Note*: The System.Nullable<T> type specifies the nonnullable value type constraint for T. Thus, recursively constructed types of the forms T?? and Nullable<Nullable<T>> are prohibited. *end note*]

Pointer types are never allowed to be type arguments and are not considered to satisfy either the reference type or value type constraints.

If a constraint is a class type, an interface type, or a type parameter, that type specifies a minimal "base type" that every type argument used for that type parameter shall support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile-time. The type argument supplied shall satisfy the conditions described in §9.4.5.

A *class-type* constraint shall satisfy the following rules:

- The type shall be a class type.
- The type shall not be sealed.
- The type shall not be one of the following types: System.Array, System.Delegate, System.Enum, or System.ValueType.
- The type shall not be object.
- At most one constraint for a given type parameter may be a class type.

A type specified as an *interface-type* constraint shall satisfy the following rules:

- The type shall be an interface type.
- A type shall not be specified more than once in a given where clause.

In either case, the constraint may involve any of the type parameters of the associated type or method declaration as part of a constructed type, and may involve the type being declared.

Any class or interface type specified as a type parameter constraint shall be at least as accessible (§8.5.5) as the generic type or method being declared.

A type specified as a *type-parameter* constraint shall satisfy the following rules:

- The type shall be a type parameter.
- A type shall not be specified more than once in a given where clause.

In addition there shall be no cycles in the dependency graph of type parameters, where dependency is a transitive relation defined by:

- If a type parameter T is used as a constraint for type parameter S then S *depends on* T.
- If a type parameter S depends on a type parameter T and T depends on a type parameter U then S *depends on* U.

Given this relation, it is a compile-time error for a type parameter to depend on itself (directly or indirectly).

Any constraints shall be consistent among dependent type parameters. If type parameter S depends on type parameter T then:

- T shall not have the value type constraint. Otherwise, T is effectively sealed so S would be forced to be the same type as T, eliminating the need for two type parameters.
- If S has the value type constraint then T shall not have a *class-type* constraint.
- If S has a *class-type* constraint A and T has a *class-type* constraint B then there shall be an identity conversion or implicit reference conversion from A to B or an implicit reference conversion from B to A.

• If S also depends on type parameter U and U has a *class-type* constraint A and T has a *class-type* constraint B then there shall be an identity conversion or implicit reference conversion from A to B or an implicit reference conversion from B to A.

It is valid for S to have the value type constraint and T to have the reference type constraint. Effectively this limits T to the types System.Object, System.ValueType, System.Enum, and any interface type.

If the where clause for a type parameter includes a constructor constraint (which has the form new()), it is possible to use the new operator to create instances of the type (§12.7.11.2). Any type argument used for a type parameter with a constructor constraint shall be a value type, a non-abstract class having a public parameterless constructor, or a type parameter having the value type constraint or constructor constraint.

[*Example*: The following are examples of constraints:

```
interface IPrintable
{
   void Print();
}
interface IComparable<T>
Ł
   int CompareTo(T value);
}
interface IKeyProvider<T>
   T GetKey();
}
class Printer<T> where T: IPrintable {...}
class SortedList<T> where T: IComparable<T> {...}
class Dictionary<K,V>
   where K: IComparable<K>
where V: IPrintable, IKeyProvider<K>, new()
{
   ...
}
```

The following example is in error because it causes a circularity in the dependency graph of the type parameters:

```
class Circular<S,T>
   where S: T
   where T: S // Error, circularity in dependency graph
{
    ...
}
```

The following examples illustrate additional invalid situations:

```
class Sealed<S,T>
   where S: T
   where T: struct // Error, T is sealed
{
    ...
}
class A {...}
class A {...}
class Incompat<S,T>
   where S: A, T
   where T: B // Error, incompatible class-type constraints
{
    ...
}
```

```
class StructWithClass<S,T,U>
   where S: struct, T
   where T: U
   where U: A // Error, A incompatible with struct
{
    ...
}
```

end example]

The *dynamic erasure* of a type C is type  $C_0$  constructed as follows:

- If C is a nested type Outer. Inner then C<sub>0</sub> is a nested type Outer<sub>0</sub>. Inner<sub>0</sub>.
- If C is a constructed type G<A<sup>1</sup>, ..., A<sup>n</sup>> with type arguments A<sup>1</sup>, ..., A<sup>n</sup> then C<sub>0</sub> is the constructed type G<A<sup>1</sup><sub>0</sub>, ..., A<sup>n</sup><sub>0</sub>>.
- If C is an *array type* E[] then C<sub>0</sub> is the array type E<sub>0</sub>[].
- If C is a *pointer type*  $E^*$  then  $C_0$  is the pointer type  $E_0^*$ .
- If C is dynamic then  $C_0$  is object.
- Otherwise, C<sub>0</sub> is C.

The *effective base class* of a type parameter T is defined as follows:

Let R be a set of types such that:

- For each constraint of T that is a *type-parameter*, R contains its effective base class.
- For each constraint of T that is a *struct-type*, R contains System.ValueType.
- For each constraint of T that is an *enumeration type*, R contains System. Enum.
- For each constraint of T that is a *delegate type*, R contains its dynamic erasure.
- For each constraint of T that is an *array type*, R contains System.Array.
- For each constraint of T that is a *class-type*, R contains its dynamic erasure.

#### Then

- If T has the value type constraint, its *effective base class* is System.ValueType.
- Otherwise, if R is empty then the *effective base class* is object.
- Otherwise, the *effective base class* of T is the most-encompassed type (§11.5.3) of set R. If the set has no encompassed type, the *effective base class* of T is object. The consistency rules ensure that the most-encompassed type exists.

If the type parameter is a method type parameter whose constraints are inherited from the base method the *effective base class* is calculated after type substitution.

These rules ensure that the effective base class is always a *class-type*.

The *effective interface set* of a type parameter T is defined as follows:

- If T has no secondary-constraints, its effective interface set is empty.
- If T has *interface-type* constraints but no *type-parameter* constraints, its effective interface set is the set of dynamic erasures of its *interface-type* constraints.
- If T has no *interface-type* constraints but has *type-parameter* constraints, its effective interface set is the union of the effective interface sets of its *type-parameter* constraints.

• If T has both *interface-type* constraints and *type-parameter* constraints, its effective interface set is the union of the set of dynamic erasures of its *interface-type* constraints and the effective interface sets of its *type-parameter* constraints.

A type parameter is *known to be a reference type* if it has the reference type constraint or its effective base class is not object or System.ValueType.

Values of a constrained type parameter type can be used to access the instance members implied by the constraints. [*Example*: In the following:

```
interface IPrintable
{
    void Print();
}
class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

the methods of IPrintable can be invoked directly on x because T is constrained to always implement IPrintable. *end example*]

When a partial generic type declaration includes constraints, the constraints shall agree with all other parts that include constraints. Specifically, each part that includes constraints shall have constraints for the same set of type parameters, and for each type parameter, the sets of primary, secondary, and constructor constraints shall be equivalent. Two sets of constraints are equivalent if they contain the same members. If no part of a partial generic type specifies type parameter constraints, the type parameters are considered unconstrained. [*Example*:

is correct because those parts that include constraints (the first two) effectively specify the same set of primary, secondary, and constructor constraints for the same set of type parameters, respectively. *end example*]

# 15.2.6 Class body

The *class-body* of a class defines the members of that class.

class-body: { class-member-declarations<sub>opt</sub> }

## 15.2.7 Partial declarations

The modifier partial is used when defining a class, struct, or interface type in multiple parts. The partial modifier is a contextual keyword (§7.4.4) and only has special meaning immediately before one of the keywords class, struct, or interface.

Each part of a *partial type* declaration shall include a partial modifier and shall be declared in the same namespace or containing type as the other parts. The partial modifier indicates that additional parts of the type declaration might exist elsewhere, but the existence of such additional parts is not a requirement; it is valid for the only declaration of a type to include the partial modifier.

All parts of a partial type shall be compiled together such that the parts can be merged at compile-time. Partial types specifically do not allow already compiled types to be extended.

Nested types can be declared in multiple parts by using the partial modifier. Typically, the containing type is declared using partial as well, and each part of the nested type is declared in a different part of the containing type.

[*Example*: The following partial class is implemented in two parts, which reside in different source files. The first part is machine generated by a database-mapping tool while the second part is manually authored:

When the two parts above are compiled together, the resulting code behaves as if the class had been written as a single unit, as follows:

```
end example]
```

The handling of attributes specified on the type or type parameters of different parts of a partial declaration is discussed in §22.3.

# 15.3 Class members

# 15.3.1 General

The members of a class consist of the members introduced by its *class-member-declarations* and the members inherited from the direct base class.

class-member-declarations: class-member-declaration class-member-declarations class-member-declaration

class-member-declaration: constant-declaration field-declaration method-declaration property-declaration event-declaration indexer-declaration operator-declaration constructor-declaration finalizer-declaration static-constructor-declaration type-declaration

The members of a class are divided into the following categories:

- Constants, which represent constant values associated with the class (§15.4).
- Fields, which are the variables of the class (§15.5).
- Methods, which implement the computations and actions that can be performed by the class (§15.6).
- Properties, which define named characteristics and the actions associated with reading and writing those characteristics (§15.7).
- Events, which define notifications that can be generated by the class (§15.8).
- Indexers, which permit instances of the class to be indexed in the same way (syntactically) as arrays (§15.9).
- Operators, which define the expression operators that can be applied to instances of the class (§15.10).
- Instance constructors, which implement the actions required to initialize instances of the class (§15.11)
- Finalizers, which implement the actions to be performed before instances of the class are permanently discarded (§15.13).
- Static constructors, which implement the actions required to initialize the class itself (§15.12).
- Types, which represent the types that are local to the class (§14.7).

Members that can contain executable code are collectively known as the *function members* of the class. The function members of a class are the methods, properties, events, indexers, operators, instance constructors, finalizers, and static constructors of that class.

A *class-declaration* creates a new declaration space (§8.3), and the *type-parameters* and the *class-member-declarations* immediately contained by the *class-declaration* introduce new members into this declaration space. The following rules apply to *class-member-declarations*:

- Instance constructors, finalizers, and static constructors shall have the same name as the immediately enclosing class. All other members shall have names that differ from the name of the immediately enclosing class.
- The name of a type parameter in the *type-parameter-list* of a class declaration shall differ from the names of all other type parameters in the same *type-parameter-list* and shall differ from the name of the class and the names of all members of the class.
- The name of a type shall differ from the names of all non-type members declared in the same class. If two or more type declarations share the same fully qualified name, the declarations shall have the

partial modifier (§15.2.7) and these declarations combine to define a single type. [*Note*: Since the fully qualified name of a type declaration encodes the number of type parameters, two distinct types may share the same name as long as they have different number of type parameters. *end note*]

- The name of a constant, field, property, or event shall differ from the names of all other members declared in the same class.
- The name of a method shall differ from the names of all other non-methods declared in the same class. In addition, the signature (§8.6) of a method shall differ from the signatures of all other methods declared in the same class, and two methods declared in the same class shall not have signatures that differ solely by ref and out.
- The signature of an instance constructor shall differ from the signatures of all other instance constructors declared in the same class, and two constructors declared in the same class shall not have signatures that differ solely by ref and out.
- The signature of an indexer shall differ from the signatures of all other indexers declared in the same class.
- The signature of an operator shall differ from the signatures of all other operators declared in the same class.

The inherited members of a class (§15.3.4) are not part of the declaration space of a class. [*Note*: Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member). *end note*]

The set of members of a type declared in multiple parts (§15.2.7) is the union of the members declared in each part. The bodies of all parts of the type declaration share the same declaration space (§8.3), and the scope of each member (§8.7) extends to the bodies of all the parts. The accessibility domain of any member always includes all the parts of the enclosing type; a private member declared in one part is freely accessible from another part. It is a compile-time error to declare the same member in more than one part of the type, unless that member is a type having the partial modifier. [*Example*:

```
partial class A
   int x;
                           // Error, cannot declare x more than once
                           // Ok, Inner is a partial type
   partial class Inner
      int y;
   }
}
partial class A
                           // Error, cannot declare x more than once
   int x;
                           // Ok, Inner is a partial type
   partial class Inner
      int z;
   }
}
```

## end example]

Field initialization order can be significant within C# code, and some guarantees are provided, as defined in §15.5.6.1. Otherwise, the ordering of members within a type is rarely significant, but may be significant when interfacing with other languages and environments. In these cases, the ordering of members within a type declared in multiple parts is undefined.

## 15.3.2 The instance type

Each class declaration has an associated *instance type*. For a generic class declaration, the instance type is formed by creating a constructed type (§9.4) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it can only be used where the type parameters are in scope; that is, inside the class declaration. The instance

type is the type of this for code written inside the class declaration. For non-generic classes, the instance type is simply the declared class. [*Example*: The following shows several class declarations along with their instance types:

class A <t></t>	// instance type: A <t></t>
class B {}	// instance type: A <t>.B</t>
class C <u> {} }</u>	<pre>// instance type: A<t>.C<u></u></t></pre>
class D {}	// instance type: D

end example]

## 15.3.3 Members of constructed types

The non-inherited members of a constructed type are obtained by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the constructed type. The substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution.

[Example: Given the generic class declaration

```
class Gen<T,U>
{
   public T[,] a;
   public void G(int i, T t, Gen<U,T> gt) {...}
   public U Prop { get {...} set {...} }
   public int H(double d) {...}
}
```

the constructed type Gen<int[], IComparable<string>> has the following members:

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

The type of the member a in the generic class declaration Gen is "two-dimensional array of T", so the type of the member a in the constructed type above is "two-dimensional array of single-dimensional array of int", or int[,][]. end example]

Within instance function members, the type of this is the instance type (§15.3.2) of the containing declaration.

All members of a generic class can use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type (§9.4.3) is used at run-time, each use of a type parameter is replaced with the type argument supplied to the constructed type. [*Example*:

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;
    public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}
class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.writeLine(x1.f1); // Prints 1
```

```
C<double> x2 = new C<double>(3.1415);
Console.WriteLine(x2.f1); // Prints 3.1415
}
```

end example]

# 15.3.4 Inheritance

A class *inherits* the members of its direct base class. Inheritance means that a class implicitly contains all members of its direct base class, except for the instance constructors, finalizers, and static constructors of the base class. Some important aspects of inheritance are:

- Inheritance is transitive. If C is derived from B, and B is derived from A, then C inherits the members declared in B as well as the members declared in A.
- A derived class *extends* its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Instance constructors, finalizers, and static constructors are not inherited, but all other members are, regardless of their declared accessibility (§8.5). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
- A derived class can *hide* (§8.7.2.3) inherited members by declaring new members with the same name or signature. However, hiding an inherited member does not remove that member—it merely makes that member inaccessible directly through the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion (§11.2.7) exists from a derived class type to any of its base class types. Thus, a reference to an instance of some derived class can be treated as a reference to an instance of any of its base classes.
- A class can declare virtual methods, properties, indexers, and events, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation vary depending on the run-time type of the instance through which that function member is invoked.

The inherited members of a constructed class type are the members of the immediate base class type (§15.2.4.2), which is found by substituting the type arguments of the constructed type for each occurrence of the corresponding type parameters in the *base-class-specification*. These members, in turn, are transformed by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the *base-class-specification*. [Example:

```
class B<U>
{
   public U F(long index) {...}
}
class D<T>: B<T[]>
{
   public T G(string s) {...}
}
```

In the code above, the constructed type D<int> has a non-inherited member public int G(string s) obtained by substituting the type argument int for the type parameter T. D<int> also has an inherited member from the class declaration B. This inherited member is determined by first determining the base class type B<int[]> of D<int> by substituting int for T in the base class specification B<T[]>. Then, as a type argument to B, int[] is substituted for U in public U F(long index), yielding the inherited member public int[] F(long index). end example]

# 15.3.5 The new modifier

A *class-member-declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to *hide* the base class member. See §8.7.2.3 for a precise specification of when a member hides an inherited member.

An inherited member M is considered to be **available** if M is accessible and there is no other inherited accessible member N that already hides M. Implicitly hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning unless the declaration of the derived class member includes a new modifier to explicitly indicate that the derived member is intended to hide the base member. If one or more parts of a partial declaration (§15.2.7) of a nested type include the new modifier, no warning is issued if the nested type hides an available inherited member.

If a new modifier is included in a declaration that doesn't hide an available inherited member, a warning to that effect is issued.

# 15.3.6 Access modifiers

A class-member-declaration can have any one of the five possible kinds of declared accessibility (§8.5.2): public, protected internal, protected, internal, or private. Except for the protected internal combination, it is a compile-time error to specify more than one access modifier. When a class-member-declaration does not include any access modifiers, private is assumed.

## 15.3.7 Constituent types

Types that are used in the declaration of a member are called the *constituent types* of that member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or instance constructor. The constituent types of a member shall be at least as accessible as that member itself (§8.5.5).

## 15.3.8 Static and instance members

Members of a class are either *static members* or *instance members*. [*Note*: Generally speaking, it is useful to think of static members as belonging to classes and instance members as belonging to objects (instances of classes). *end note*]

When a field, method, property, event, operator, or constructor declaration includes a static modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member M is referenced in a *member-access* (§12.7.5) of the form E.M, E shall denote a type that has a member M. It is a compile-time error for E to denote an instance.
- A static field in a non-generic class identifies exactly one storage location. No matter how many instances of a non-generic class are created, there is only ever one copy of a static field. Each distinct closed constructed type (§9.4.3) has its own set of static fields, regardless of the number of instances of the closed constructed type.
- A static function member (method, property, event, operator, or constructor) does not operate on a specific instance, and it is a compile-time error to refer to this in such a function member.

When a field, method, property, event, indexer, constructor, or finalizer declaration does not include a static modifier, it declares an instance member. (An instance member is sometimes called a non-static member.) Instance members have the following characteristics:

- When an instance member M is referenced in a *member-access* (§12.7.5) of the form E.M, E shall denote an instance of a type that has a member M. It is a binding-time error for E to denote a type.
- Every instance of a class contains a separate set of all instance fields of the class.
- An instance function member (method, property, indexer, instance constructor, or finalizer) operates on a given instance of the class, and this instance can be accessed as this (§12.7.8).

[*Example*: The following example illustrates the rules for accessing static and instance members:

```
class Test
{
int x;
static int y;
```

```
void F() {
                      // Ok, same as this.x = 1
   x = 1;
                      // Ok, same as Test.y = 1
   y = 1;
}
static void G() {
   x = 1;
                      // Error, cannot access this.x
   y = 1;
                      // Ok, same as Test.y = 1
}
static void Main() {
   Test t = new Test();
   t.x = 1;
                  // 0k
   t.y = 1; // Error, cannot access static member through instance
Test.x = 1; // Error, cannot access instance member through type
   Test.y = 1;
                  // Ok
}
```

The F method shows that in an instance function member, a *simple-name* (§12.7.3) can be used to access both instance members and static members. The G method shows that in a static function member, it is a compile-time error to access an instance member through a *simple-name*. The Main method shows that in a *member-access* (§12.7.5), instance members shall be accessed through instances, and static members shall be accessed through types. *end example*]

## 15.3.9 Nested types

## 15.3.9.1 General

}

A type declared within a class or struct is called a **nested type**. A type that is declared within a compilation unit or namespace is called a **non-nested type**. [*Example*: In the following example:

```
using System;
class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

class B is a nested type because it is declared within class A, and class A is a non-nested type because it is declared within a compilation unit. *end example*]

## 15.3.9.2 Fully qualified name

The fully qualified name (§8.8.3) for a nested type declarationis S.N where S is the fully qualified name of the type declarationin which type N is declared and N is the unqualified name (§8.8.2) of the nested type declaration (including any *generic-dimension-specifier* (§12.7.12)).

#### 15.3.9.3 Declared accessibility

Non-nested types can have public or internal declared accessibility and have internal declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of five forms of declared accessibility (public, protected internal, protected, internal, or private) and, like other class members, defaults to private declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (public, internal, or private) and, like other struct members, defaults to private declared accessibility.

#### [Example: The example

```
public class List
    // Private data structure
    private class Node
        public object Data;
        public Node Next;
        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }
   private Node first = null;
private Node last = null;
    // Public interface
    public void AddToFront(object o) {...}
public void AddToBack(object o) {...}
   public object RemoveFromFront() {...}
public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

declares a private nested class Node. end example]

#### 15.3.9.4 Hiding

A nested type may hide (§8.7.2.2) a base member. The new modifier (§15.3.5) is permitted on nested type declarations so that hiding can be expressed explicitly. [*Example*: The example

```
using System;
class Base
{
   public static void M() {
    Console.WriteLine("Base.M");
    }
}
class Derived: Base
    new public class M
    {
        public static void F() {
    Console.WriteLine("Derived.M.F");
        }
    }
}
class Test
ł
    static void Main() {
        Derived.M.F();
}
```

shows a nested class M that hides the method M defined in Base. end example]

## 15.3.9.5 this access

A nested type and its containing type do not have a special relationship with regard to *this-access* (§12.7.8). Specifically, this within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the this for the instance of the containing type as a constructor argument for the nested type. [*Example*: The following example

using System;

```
class C
ł
   int i = 123;
public void F() {
      Nested n = new Nested(this);
      n.G();
   }
   public class Nested
      C this_c;
      public Nested(C c) {
          this_c = c;
      }
      public void G() {
          Console.WriteLine(this_c.i);
   }
}
class Test
   static void Main() {
      C c = new C();
      c.F();
   }
}
```

shows this technique. An instance of C creates an instance of Nested, and passes its own this to Nested's constructor in order to provide subsequent access to C's instance members. *end example*]

15.3.9.6 Access to private and protected members of the containing type

A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have private and protected declared accessibility. [*Example*: The example

```
using System;
class C
ł
   private static void F() {
    Console.WriteLine("C.F");
   }
   public class Nested
   ł
       public static void G() {
           F();
       }
   }
}
class Test
ł
   static void Main() {
       C.Nested.G();
   }
}
```

shows a class C that contains a nested class Nested. Within Nested, the method G calls the static method F defined in C, and F has private declared accessibility. *end example*]

A nested type also may access protected members defined in a base type of its containing type. [*Example*: In the following code

using System;

```
class Base
   protected void F() {
      Console.WriteLine("Base.F");
   }
}
class Derived: Base
   public class Nested
      public void G() {
         Derived d = new Derived();
         d.F();
                      // ok
      }
   }
}
class Test
ł
   static void Main() {
      Derived.Nested n = new Derived.Nested();
      n.G();
   }
}
```

the nested class Derived.Nested accesses the protected method F defined in Derived's base class, Base, by calling through an instance of Derived. *end example*]

15.3.9.7 Nested types in generic classes

A generic class declaration may contain nested type declarations. The type parameters of the enclosing class may be used within the nested types. A nested type declaration may contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, shall be named. However, from within the outer class, the nested type may be used without qualification; the instance type of the outer class may be implicitly used when constructing the nested type. [*Example*: The following shows three different correct ways to refer to a constructed type created from Inner; the first two are equivalent:

```
class Outer<T>
{
   class Inner<U>
   {
      public static void F(T t, U u) {...}
   }
   static void F(T t) {
      Outer<T>.Inner<string>.F(t, "abc");
Inner<string>.F(t, "abc");
                                                  // These two statements have
                                                  // the same effect
      Outer<int>.Inner<string>.F(3, "abc");
                                                  // This type is different
      Outer.Inner<string>.F(t, "abc");
                                                  // Error, Outer needs type
arg
}
```

```
end example]
```

Although it is bad programming style, a type parameter in a nested type can hide a member or type parameter declared in the outer type. [*Example*:

```
class Outer<T>
{
    class Inner<T> // Valid, hides Outer's T
    {
        public T t; // Refers to Inner's T
    }
}
```

end example]

## 15.3.10 Reserved member names

## 15.3.10.1 General

To facilitate the underlying C# run-time implementation, for each source member declaration that is a property, event, or indexer, the implementation shall reserve two method signatures based on the kind of the member declaration, its name, and its type (§15.3.10.2, §15.3.10.3, §15.3.10.4). It is a compile-time error for a program to declare a member whose signature matches a signature reserved by a member declared in the same scope, even if the underlying run-time implementation does not make use of these reservations.

The reserved names do not introduce declarations, thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance (§15.3.4), and can be hidden with the new modifier (§15.3.5).

[Note: The reservation of these names serves three purposes:

- 1. To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
- 2. To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
- 3. To help ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

#### end note]

The declaration of a finalizer (§15.13) also causes a signature to be reserved (§15.3.10.5).

15.3.10.2 Member names reserved for properties

For a property P (§15.7) of type T, the following signatures are reserved:

```
T get_P();
void set_P(T value);
```

Both signatures are reserved, even if the property is read-only or write-only.

[Example: In the following code

```
using System;
class A
{
    public int P {
        get { return 123; }
    }
}
class B: A
{
    new public int get_P() {
        return 456;
    }
    new public void set_P(int value) {
    }
}
```

```
class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

a class A defines a read-only property P, thus reserving signatures for get\_P and set\_P methods. A class B derives from A and hides both of these reserved signatures. The example produces the output:

123 123 456

end example]

15.3.10.3 Member names reserved for events

For an event E (§15.8) of delegate type T, the following signatures are reserved:

void add\_E(T handler); void remove\_E(T handler);

15.3.10.4 Member names reserved for indexers

For an indexer (§15.9) of type T with parameter-list L, the following signatures are reserved:

```
T get_Item(L);
void set_Item(L, T value);
```

Both signatures are reserved, even if the indexer is read-only or write-only.

Furthermore the member name Item is reserved.

15.3.10.5 Member names reserved for finalizers

For a class containing a finalizer (§15.13), the following signature is reserved:

void Finalize();

# 15.4 Constants

A *constant* is a class member that represents a constant value: a value that can be computed at compiletime. A *constant-declaration* introduces one or more constants of a given type.

```
constant-declaration:
```

attributes<sub>opt</sub> constant-modifiers<sub>opt</sub> const type constant-declarators ;

```
constant-modifiers:
constant-modifier
constant-modifiers constant-modifier
```

constant-modifier: new public protected internal private constant-declarators: constant-declarator constant-declarators , constant-declarator constant-declarator: identifier = constant-expression A constant-declaration may include a set of attributes (§22), a new modifier (§15.3.5), and a valid combination of the four access modifiers (§15.3.6). The attributes and modifiers apply to all of the members declared by the constant-declaration. Even though constants are considered static members, a constant-declaration neither requires nor allows a static modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant-declarators*, each of which introduces a new member. A *constant-declarator* consists of an *identifier* that names the member, followed by an "=" token, followed by a *constant-expression* (§12.20) that gives the value of the member.

The *type* specified in a constant declaration shall be sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, an *enum-type*, or a *reference-type*. Each *constant-expression* shall yield a value of the target type or of a type that can be converted to the target type by an implicit conversion (§11.2).

The *type* of a constant shall be at least as accessible as the constant itself (§8.5.5).

The value of a constant is obtained in an expression using a *simple-name* (§12.7.3) or a *member-access* (§12.7.5).

A constant can itself participate in a *constant-expression*. Thus, a constant may be used in any construct that requires a *constant-expression*. [*Note*: Examples of such constructs include case labels, goto case statements, enum member declarations, attributes, and other constant declarations. *end note*]

[Note: As described in §12.20, a constant-expression is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a *reference-type* other than string is to apply the new operator, and since the new operator is not permitted in a constant-expression, the only possible value for constants of *reference-types* other than string is null. *end note*]

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile-time by a *constant-expression*, a readonly field (§15.5.3) may be used instead. [*Note*: The versioning semantics of const and readonly differ (§15.5.3.3). *end-note*]

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. [*Example*:

```
class A
{
    public const double x = 1.0, Y = 2.0, Z = 3.0;
}
```

is equivalent to

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

#### end example]

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. [*Example*: In the following code

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}
```

```
class B
{
    public const int Z = A.Y + 1;
}
```

the compiler first evaluates A.Y, then evaluates B.Z, and finally evaluates A.X, producing the values 10, 11, and 12. *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*]

# 15.5 Fields

# 15.5.1 General

A *field* is a member that represents a variable associated with an object or class. A *field-declaration* introduces one or more fields of a given type.

```
field-declaration:
    attributes<sub>opt</sub> field-modifiers<sub>opt</sub> type variable-declarators ;
field-modifiers:
    field-modifier
    field-modifiers field-modifier
field-modifier:
    new
    public
    protected
    internal
    private
    static
    readonly
    volatile
variable-declarators:
    variable-declarator
    variable-declarators, variable-declarator
variable-declarator:
    identifier
    identifier = variable-initializer
variable-initializer:
    expression
    array-initializer
```

A *field-declaration* may include a set of *attributes* (§22), a new modifier (§15.3.5), a valid combination of the four access modifiers (§15.3.6), and a static modifier (§15.5.2). In addition, a *field-declaration* may include a readonly modifier (§15.5.3) or a volatile modifier (§15.5.4), but not both. The attributes and modifiers apply to all of the members declared by the *field-declaration*. It is an error for the same modifier to appear multiple times in a *field declaration*.

The *type* of a *field-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable-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* (§15.5.6) that gives the initial value of that member.

The type of a field shall be at least as accessible as the field itself (§8.5.5).

The value of a field is obtained in an expression using a *simple-name* (§12.7.3), a *member-access* (§12.7.5) or a base-access (§12.7.9). The value of a non-readonly field is modified using an *assignment* (§12.18). The

value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators (§12.7.10) and prefix increment and decrement operators (§12.8.6).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. [*Example*:

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

is equivalent to

```
class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

end example]

## 15.5.2 Static and instance fields

When a field declaration includes a static modifier, the fields introduced by the declaration are *static fields*. When no static modifier is present, the fields introduced by the declaration are *instance fields*. Static fields and instance fields are two of the several kinds of variables (§10) supported by C#, and at times they are referred to as *static variables* and *instance variables*, respectively.

As explained in §15.3.8, each instance of a class contains a complete set of the instance fields of the class, while there is only one set of static fields for each non-generic class or closed constructed type, regardless of the number of instances of the class or closed constructed type.

## 15.5.3 Readonly fields

## 15.5.3.1 General

When a *field-declaration* includes a readonly modifier, the fields introduced by the declaration are *readonly fields*. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a readonly field are permitted only in the following contexts:

- In the variable-declarator that introduces the field (by including a variable-initializer in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a readonly field as an out or ref parameter.

Attempting to assign to a readonly field or pass it as an out or ref parameter in any other context is a compile-time error.

## 15.5.3.2 Using static readonly fields for constants

A static readonly field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a const declaration, or when the value cannot be computed at compile-time. [*Example*: In the following code

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

}

and

```
private byte red, green, blue;
public Color(byte r, byte g, byte b) {
   red = r;
   green = g;
   blue = b;
}
```

the Black, White, Red, Green, and Blue members cannot be declared as const members because their values cannot be computed at compile-time. However, declaring them static readonly instead has much the same effect. *end example*]

## 15.5.3.3 Versioning of constants and static readonly fields

Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time. [*Example*: Consider an application that consists of two separate programs:

```
namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}
using System;
namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}
```

The Program1 and Program2 namespaces denote two programs that are compiled separately. Because Program1.Utils.X is declared as a static readonly field, the value output by the Console.WriteLine statement is not known at compile-time, but rather is obtained at run-time. Thus, if the value of X is changed and Program1 is recompiled, the Console.WriteLine statement will output the new value even if Program2 isn't recompiled. However, had X been a constant, the value of X would have been obtained at the time Program2 was compiled, and would remain unaffected by changes in Program1 until Program2 is recompiled. *end example*]

# 15.5.4 Volatile fields

When a *field-declaration* includes a volatile modifier, the fields introduced by that declaration are **volatile fields**. For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded programs that access fields without synchronization such as that provided by the *lock-statement* (§13.13). These optimizations can be performed by the compiler, by the run-time system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a *volatile read*. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a **volatile write**. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field shall be one of the following:

- A reference-type.
- A type-parameter that is known to be a reference type (§15.2.5).
- The type byte, sbyte, short, ushort, int, uint, char, float, bool, System.IntPtr, or System.UIntPtr.
- An enum-type having an enum base type of byte, sbyte, short, ushort, int, or uint.

[Example: The example

```
using System;
using System.Threading;
class Test
   public static int result;
   public static volatile bool finished;
   static void Thread2() {
    result = 143;
       finished = true;
   }
   static void Main() {
       finished = false;
// Run Thread2() in a new thread
       new Thread(new ThreadStart(Thread2)).Start();
       // Wait for Thread2 to signal that it has a result by setting
        / finished to true.
       for (;;) {
if (finished)<sub>.</sub>{
              Console.WriteLine("result = {0}", result);
              return;
          }
       }
   }
}
```

produces the output:

result = 143

In this example, the method Main starts a new thread that runs the method Thread2. This method stores a value into a non-volatile field called result, then stores true in the volatile field finished. The main thread waits for the field finished to be set to true, then reads the field result. Since finished has been declared volatile, the main thread shall read the value 143 from the field result. If the field finished had not been declared volatile, then it would be permissible for the store to result to be visible to the main thread *after* the store to finished, and hence for the main thread to read the value 0 from the field result. Declaring finished as a volatile field prevents any such inconsistency. *end example*]

## 15.5.5 Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (§10.3) of the field's type. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized". [*Example*: The example

```
using System;
class Test
{
static bool b;
int i;
```

```
static void Main() {
    Test t = new Test();
    Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

produces the output

b = False, i = 0

because b and i are both automatically initialized to default values. end example]

#### 15.5.6 Variable initializers

#### 15.5.6.1 General

Field declarations may include *variable-initializers*. For static fields, variable initializers correspond to assignment statements that are executed during class initialization. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

[Example: The example

```
using System;
class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";
    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

produces the output

x = 1.4142135623731, i = 100, s = Hello

because an assignment to x occurs when static field initializers execute and assignments to i and s occur when the instance field initializers execute. *end example*]

The default value initialization described in §15.5.5 occurs for all fields, including fields that have variable initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields in that instance are first initialized to their default values, and then the instance field initializers are executed in textual order. When there are field declarations in multiple partial type declarations for the same type, the order of the parts is unspecified. However, within each part the field initializers are executed in order.

It is possible for static fields with variable initializers to be observed in their default value state. [*Example*: However, this is strongly discouraged as a matter of style. The example

```
using System;
class Test
{
  static int a = b + 1;
  static int b = a + 1;
  static void Main() {
     Console.WriteLine("a = {0}, b = {1}", a, b);
  }
}
```

exhibits this behavior. Despite the circular definitions of a and b, the program is valid. It results in the output

a = 1, b = 2

because the static fields a and b are initialized to 0 (the default value for int) before their initializers are executed. When the initializer for a runs, the value of b is zero, and so a is initialized to 1. When the initializer for b runs, the value of a is already 1, and so b is initialized to 2. *end example*]

#### 15.5.6.2 Static field initialization

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration (§15.5.6.1). Within a partial class, the meaning of "textual order" is specified by §15.5.6.1. If a static constructor (§15.12) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. [*Example*: The example

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine("{0} {1}", B.Y, A.X);
   public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}
class A
{
   public static int X = Test.F("Init A");
}
class B
{
   public static int Y = Test.F("Init B");
}
```

might produce either the output:

Init A Init B 1 1

or the output:

Init B Init A 1 1

because the execution of X's initializer and Y's initializer could occur in either order; they are only constrained to occur before the references to those fields. However, in the example:

```
using System;
class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }
    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}
class A
{
    static A() {}
    public static int X = Test.F("Init A");
}
```

```
class B
{
    static B() {}
    public static int Y = Test.F("Init B");
}
```

the output shall be:

Init B Init A 1 1

because the rules for when static constructors execute (as defined in §15.12) provide that B's static constructor (and hence B's static field initializers) shall run before A's static constructor and field initializers. *end example*]

# 15.5.6.3 Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (§15.11.3) of that class. Within a partial class, the meaning of "textual order" is specified by §15.5.6.1. The variable initializers are executed in the textual order in which they appear in the class declaration (§15.5.6.1). The class instance creation and initialization process is described further in §15.11.

A variable initializer for an instance field cannot reference the instance being created. Thus, it is a compiletime error to reference this in a variable initializer, as it is a compile-time error for a variable initializer to reference any instance member through a *simple-name*. [*Example*: In the following code

```
class A
{
    int x = 1;
    int y = x + 1; // Error, reference to instance member of this
}
```

the variable initializer for y results in a compile-time error because it references a member of the instance being created. *end example*]

# 15.6 Methods

# 15.6.1 General

A *method* is a member that implements a computation or action that can be performed by an object or class. Methods are declared using *method-declarations*:

method-declaration: method-header method-body

method-header:

attributes<sub>opt</sub> method-modifiers<sub>opt</sub> partial<sub>opt</sub> return-type member-name type-parameter-list<sub>opt</sub>

( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub>

method-modifiers: method-modifier

method-modifiers method-modifier

```
method-modifier:
   new
   public
   protected
   internal
   private
   static
   virtual
   sealed
   override
   abstract
   extern
   async
return-type:
   type
   void
member-name:
   identifier
   interface-type . identifier
method-body:
   block
   ;
```

A *method-declaration* may include a set of *attributes* (§22) and a valid combination of the four access modifiers (§15.3.6), the new (§15.3.5), static (§15.6.3), virtual (§15.6.4), override (§15.6.5), sealed (§15.6.6), abstract (§15.6.7), extern (§15.6.8) and async (§15.15) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

- The declaration includes a valid combination of access modifiers (§15.3.6).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: static, virtual, and override.
- The declaration includes at most one of the following modifiers: new and override.
- If the declaration includes the abstract modifier, then the declaration does not include any of the following modifiers: static, virtual, sealed, or extern.
- If the declaration includes the private modifier, then the declaration does not include any of the following modifiers: virtual, override, or abstract.
- If the declaration includes the sealed modifier, then the declaration also includes the override modifier.
- If the declaration includes the partial modifier, then it does not include any of the following modifiers: new, public, protected, internal, private, virtual, sealed, override, abstract, or extern.

The *return-type* of a method declaration specifies the type of the value computed and returned by the method. The *return-type* is void if the method does not return a value. If the declaration includes the partial modifier, then the return type shall be void.

A generic method is a method whose declaration includes a *type-parameter-list*. This specifies the type parameters for the method. The optional *type-parameter-constraints-clauses* specify the constraints for the type parameters. A *method-declaration* shall not have *type-parameter-constraints-clauses* unless it also has a *type-parameter-list*. A *method-declaration* for an explicit interface member implementation shall not have any *type-parameter-constraints-clauses*. A generic *method-declaration* for an explicit interface member implementation shall not have any *type-parameter-constraints-clauses*. A generic *method-declaration* for an explicit interface member implementation inherits any constraints from the constraints on the interface method. Similarly, a method declaration with the override modifier shall not have any *type-parameter-constraints-clauses* 

and the constraints of the method's type parameters are inherited from the virtual method being overridden. The *member-name* specifies the name of the method. Unless the method is an explicit interface member implementation (§18.6.2), the *member-name* is simply an *identifier*. For an explicit interface member implementation, the *member-name* consists of an *interface-type* followed by a "." and an *identifier*. In this case, the declaration shall include no modifiers other than (possibly) extern or async.

The optional formal-parameter-list specifies the parameters of the method (§15.6.2).

The *return-type* and each of the types referenced in the *formal-parameter-list* of a method shall be at least as accessible as the method itself (§8.5.5).

For abstract and extern methods, the *method-body* consists simply of a semicolon. For partial methods the *method-body* may consist of either a semicolon or a *block*. For all other methods, the *method-body* consists of a *block*, which specifies the statements to execute when the method is invoked.

If the *method-body* consists of a semicolon, the declaration shall not include the async modifier.

The name, the number of type parameters, and the formal parameter list of a method define the signature (§8.6) of the method. Specifically, the signature of a method consists of its name, the number of its type parameters, and the number, *parameter-mode-modifiers* (§15.6.2.1), and types of its formal parameters. The return type is not part of a method's signature, nor are the names of the formal parameters, the names of the type parameters, or the constraints. When a formal parameter type references a type parameter of the method, the ordinal position of the type parameter (not the name of the type parameter) is used for type equivalence.

The name of a method shall differ from the names of all other non-methods declared in the same class. In addition, the signature of a method shall differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by ref and out.

The method's *type-parameters* are in scope throughout the *method-declaration*, and can be used to form types throughout that scope in *return-type*, *method-body*, and *type-parameter-constraints-clauses* but not in *attributes*.

All formal parameters and type parameters shall have different names.

# 15.6.2 Method parameters

## 15.6.2.1 General

The parameters of a method, if any, are declared by the method's formal-parameter-list.

```
formal-parameter-list:
    fixed-parameters
    fixed-parameters
    fixed-parameters
    fixed-parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters , fixed-parameter

fixed-parameter:
    attributes<sub>opt</sub> parameter-modifier<sub>opt</sub> type identifier default-argument<sub>opt</sub>

default-argument:
    = expression

parameter-modifier:
    parameter-modifier
    this
```

```
parameter-mode-modifier:
    ref
    out
parameter-array:
    attributes<sub>opt</sub> 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* (§22); an optional ref, out, or this modifier; a *type*; an *identifier*; and an optional *default-argument*. Each *fixed-parameter* declares a parameter of the given type with the given name. The this modifier designates the method as an extension method and is only allowed on the first parameter of a static method in a non-generic, non-nested static class. Extension methods are further described in §15.6.10. A *fixed-parameter* with a *default-argument* is known as an *optional parameter*, whereas a *fixed-parameter* without a *default-argument* is a *required parameter*. A required parameter may not appear after an optional parameter in a *formal-parameter-list*.

A parameter with a ref, out or this modifier cannot have a *default-argument*. The *expression* in a *default-argument* shall be one of the following:

- a constant-expression
- an expression of the form new S() where S is a value type
- an expression of the form default(S) where S is a value type

The *expression* shall be implicitly convertible by an identity or nullable conversion to the type of the parameter.

If optional parameters occur in an implementing partial method declaration (§15.6.9), an explicit interface member implementation (§18.6.2), a single-parameter indexer declaration (§15.9), or in an operator declaration (§15.10.1) the compiler should give a warning, since these members can never be invoked in a way that permits arguments to be omitted.

A parameter-array consists of an optional set of attributes (§22), 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 shall be a single-dimensional array type (§17.2). 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 §15.6.2.5.

A *parameter-array* may occur after an optional parameter, but cannot have a default value – the omission of arguments for a *parameter-array* would instead result in the creation of an empty array.

[*Example*: The following illustrates different kinds of parameters:

```
public void M(
ref int
decimal
                   i,
                    d,
    boo1
                   b = false,
   bool?
                   n = false,
                         "Hello"
   string
                   s =
   objecť
                   ο
                     = null,
                    t = default(T),
    т
   params int[] a
  { }
)
```

In the *formal-parameter-list* for M, i is a required ref parameter, d is a required value parameter, b, s, o and t are optional value parameters and a is a parameter array. *end example*]

A method declaration creates a separate declaration space (§8.3) for parameters and type parameters. Names are introduced into this declaration space by the type parameter list and the formal parameter list of the method. The body of the method, if any, is considered to be nested within this declaration space. It is an error for two members of a method declaration space to have the same name. It is an error for the method declaration space and the local variable declaration space of a nested declaration space to contain elements with the same name.

A method invocation (§12.7.6.2) 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 (§12.7.3).

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 §8.6, the ref and out modifiers are part of a method's signature, but the params modifier is not. *end note*]

#### 15.6.2.2 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 shall be an expression that is implicitly convertible (§11.2) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter—they have no effect on the actual argument given in the method invocation.

## 15.6.2.3 Reference parameters

A parameter declared with a ref modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation shall consist of the keyword ref followed by a *variable-reference* (§10.5) of the same type as the formal parameter. A variable shall be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

A method declared as an iterator (§15.14) may not have reference parameters.

#### [Example: The example

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.writeLine("i = {0}, j = {1}", i, j);
    }
}
```

produces the output

i = 2, j = 1

For the invocation of Swap in Main, x represents i and y represents j. Thus, the invocation has the effect of swapping the values of i and j. *end example*]

In a method that takes reference parameters, it is possible for multiple names to represent the same storage location. [*Example*: In the following code

```
class A
{
    string s;
    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }
    void G() {
        F(ref s, ref s);
    }
}
```

the invocation of F in G passes a reference to s for both a and b. Thus, for that invocation, the names s, a, and b all refer to the same storage location, and the three assignments all modify the instance field s. *end example*]

#### 15.6.2.4 Output parameters

A parameter declared with an out modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation shall consist of the keyword out followed by a *variable-reference* (§10.5) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and shall be definitely assigned before its value is used.

Every output parameter of a method shall be definitely assigned before the method returns.

A method declared as a partial method (§15.6.9) or an iterator (§15.14) may not have output parameters.

Output parameters are typically used in methods that produce multiple return values. [Example:

```
using System;
class Test
{
  static void SplitPath(string path, out string dir, out string name) {
    int i = path.Length;
    while (i > 0) {
        char ch = path[i - 1];
        if (ch == '\\' || ch == '/' || ch == ':') break;
        i--;
    }
    dir = path.Substring(0, i);
    name = path.Substring(i);
  }
```

```
static void Main() {
    string dir, name;
    SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
    Console.WriteLine(dir);
    Console.WriteLine(name);
  }
}
```

The example produces the output:

c:\Windows\System\
hello.txt

Note that the dir and name variables can be unassigned before they are passed to SplitPath, and that they are considered definitely assigned following the call. *end example*]

## 15.6.2.5 Parameter arrays

A parameter declared with a params modifier is a parameter array. If a formal parameter list includes a parameter array, it shall be the last parameter in the list and it shall be of a single-dimensional array type. [*Example*: The types string[] and string[][] can be used as the type of a parameter array, but the type string[,] can not. *end example*] It is not possible to combine the params modifier with the modifiers ref and out.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression that is implicitly convertible (§11.2) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each
  argument is an expression that is implicitly convertible (§11.2) to the element type of the parameter
  array. In this case, the invocation creates an instance of the parameter array type with a length
  corresponding to the number of arguments, initializes the elements of the array instance with the
  given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter (§15.6.2.2) of the same type.

[Example: The example

```
using System;
class Test
{
    static void F(params int[] args) {
        Console.write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.write(" {0}", i);
        Console.writeLine();
    }
    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

produces the output

Array contains 3 elements: 1 2 3 Array contains 4 elements: 10 20 30 40 Array contains 0 elements:

The first invocation of F simply passes the array arr as a value parameter. The second invocation of F automatically creates a four-element int[] with the given element values and passes that array instance

as a value parameter. Likewise, the third invocation of F creates a zero-element int[] and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

F(new int[] {10, 20, 30, 40});
F(new int[] {});

#### end example]

When performing overload resolution, a method with a parameter array might be applicable, either in its normal form or in its expanded form (§12.6.4.2). The expanded form of a method is available only if the normal form of the method is not applicable and only if an applicable method with the same signature as the expanded form is not already declared in the same type.

[Example: The example

```
using System;
class Test
ł
    static void F(params object[] a) {
   Console.WriteLine("F(object[])");
    }
    static void F() {
         Console.WriteLine("F()");
    }
    static void F(object a0, object a1) {
    Console.WriteLine("F(object,object)");
    }
    static void Main() {
         F();
         F(1);
         F(1, 2);
F(1, 2, 3);
F(1, 2, 3,
    }
}
```

produces the output

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so, it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked. *end example*]

When the type of a parameter array is object[], a potential ambiguity arises between the normal form of the method and the expanded form for a single object parameter. The reason for the ambiguity is that an object[] is itself implicitly convertible to type object. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

[Example: The example

using System;

```
class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.write(o.GetType().FullName);
            Console.write(" ");
        }
        Console.writeLine();
    }
    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

produces the output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of F, the normal form of F is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type object[]). Thus, overload resolution selects the normal form of F, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of F is not applicable because no implicit conversion exists from the argument type to the parameter type (type object cannot be implicitly converted to type object[]). However, the expanded form of F is applicable, so it is selected by overload resolution. As a result, a one-element object[] is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an object[]). *end example*]

## 15.6.3 Static and instance methods

When a method declaration includes a static modifier, that method is said to be a static method. When no static modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to this in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as this (§12.7.8).

The differences between static and instance members are discussed further in §15.3.8.

## 15.6.4 Virtual methods

When an instance method declaration includes a virtual modifier, that method is said to be a *virtual method*. When no virtual modifier is present, the method is said to be a *non-virtual method*.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as **overriding** that method (§15.6.5).

In a virtual method invocation, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the *compile-time type* of the instance is the determining factor. In precise terms, when a method named N is invoked

with an argument list A on an instance with a compile-time type C and a run-time type R (where R is either C or a class derived from C), the invocation is processed as follows:

- At binding-time, overload resolution is applied to C, N, and A, to select a specific method M from the set of methods declared in and inherited by C. This is described in §12.7.6.2.
- Then at run-time:
  - If M is a non-virtual method, M is invoked.
  - Otherwise, M is a virtual method, and the most derived implementation of M with respect to R is invoked.

For every virtual method declared in or inherited by a class, there exists a **most derived implementation** of the method with respect to that class. The most derived implementation of a virtual method M with respect to a class R is determined as follows:

- If R contains the introducing virtual declaration of M, then this is the most derived implementation of M with respect to R.
- Otherwise, if R contains an override of M, then this is the most derived implementation of M with respect to R.
- Otherwise, the most derived implementation of M with respect to R is the same as the most derived implementation of M with respect to the direct base class of R.

[Example: The following example illustrates the differences between virtual and non-virtual methods:

```
using System;
class A
{
    public void F() { Console.WriteLine("A.F"); }
public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
    new public void F() { Console.WriteLine("B.F"); }
public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
    static void Main() {
         B b = new B();
         A a = b;
         a.F();
b.F();
         a.G();
         b.G();
    }
}
```

In the example, A introduces a non-virtual method F and a virtual method G. The class B introduces a *new* non-virtual method F, thus *hiding* the inherited F, and also *overrides* the inherited method G. The example produces the output:

A.F B.F B.G B.G

Notice that the statement a.G() invokes B.G, not A.G. This is because the run-time type of the instance (which is B), not the compile-time type of the instance (which is A), determines the actual method implementation to invoke. *end example*]

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. [*Example*: In the following code

```
using System;
class A
{
   public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
   public override void F() { Console.WriteLine("B.F"); }
}
class C: B
   new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
   public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
   static void Main() {
      D d = new D();
      A a = d;
      B b = d;
      C c = d;
      a.F();
b.F();
c.F();
      d.F();
   }
}
```

the C and D classes contain two virtual methods with the same signature: The one introduced by A and the one introduced by C. The method introduced by C hides the method inherited from A. Thus, the override declaration in D overrides the method introduced by C, and it is not possible for D to override the method introduced by A. The example produces the output:

B.F B.F D.F D.F

Note that it is possible to invoke the hidden virtual method by accessing an instance of D through a less derived type in which the method is not hidden. *end example*]

# 15.6.5 Override methods

When an instance method declaration includes an override modifier, the method is said to be an **override method**. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration *introduces* a new method, an override method declaration *specializes* an existing inherited virtual method by providing a new implementation of that method.

The method overridden by an override declaration is known as the **overridden base method** For an override method M declared in a class C, the overridden base method is determined by examining each base class of C, starting with the direct base class of C and continuing with each successive direct base class, until in a given base class type at least one accessible method is located which has the same signature as M after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal and declared in the same program as C.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- There is exactly one such overridden base method. This restriction has effect only if the base class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- There is an identity conversion between the return type of the overridden base method and the override method.
- The override declaration and the overridden base method have the same declared accessibility. In
  other words, an override declaration cannot change the accessibility of the virtual method. However, if
  the overridden base method is protected internal and it is declared in a different assembly than the
  assembly containing the override declaration then the override declaration's declared accessibility
  shall be protected.
- The override declaration does not specify type-parameter-constraints-clauses. Instead, the constraints are inherited from the overridden base method. Constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not valid when explicitly specified, such as value types or sealed types.

[Example: The following demonstrates how the overriding rules work for generic classes:

```
abstract class C<T>
   public virtual T F() {...}
   public virtual C<T> G() {...}
   public virtual void H(C<T> x) {...}
}
class D: C<string>
   public override string F() {...}
                                            // Ok
   public override C<string> G() {...}
                                            // Ok
   public override void H(C<T> x) {...}
                                           // Error, should be C<string>
}
class E<T,U>: C<U>
   public override U F() {...}
                                            // ok
   public override C<U> G() {...}
                                            // Ok
   public override void H(C < T > x) \{ \dots \} // Error, should be C<U>
}
```

#### end example]

An override declaration can access the overridden base method using a *base-access* (§12.7.9). [*Example*: In the following code

```
class A
{
    int x;
    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}
class B: A
{
    int y;
```

```
public override void PrintFields() {
    base.PrintFields();
    Console.WriteLine("y = {0}", y);
  }
}
```

the base.PrintFields() invocation in B invokes the PrintFields method declared in A. A *base-access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in B been written ((A)this).PrintFields(), it would recursively invoke the PrintFields method declared in B, not the one declared in A, since PrintFields is virtual and the run-time type of ((A)this) is B. *end example*]

Only by including an override modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. [*Example*: In the following code

```
class A
{
   public virtual void F() {}
}
class B: A
{
   public virtual void F() {} // Warning, hiding inherited F()
}
```

the F method in B does not include an override modifier and therefore does not override the F method in A. Rather, the F method in B hides the method in A, and a warning is reported because the declaration does not include a new modifier. *end example*]

[Example: In the following code

```
class A
{
    public virtual void F() {}
}
class B: A
{
    new private void F() {} // Hides A.F within body of B
}
class C: B
{
    public override void F() {} // Ok, overrides A.F
}
```

the F method in B hides the virtual F method inherited from A. Since the new F in B has private access, its scope only includes the class body of B and does not extend to C. Therefore, the declaration of F in C is permitted to override the F inherited from A. *end example*]

# 15.6.6 Sealed methods

When an instance method declaration includes a sealed modifier, that method is said to be a *sealed method*. A sealed method overrides an inherited virtual method with the same signature. A sealed method shall also be marked with the override modifier. Use of the sealed modifier prevents a derived class from further overriding the method.

[Example: The example

```
using System;
class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
```

```
public virtual void G() {
    Console.WriteLine("A.G");
  }
}
class B: A
{
  public sealed override void F() {
    Console.WriteLine("B.F");
  }
  public override void G() {
    Console.WriteLine("B.G");
  }
}
class C: B
{
  public override void G() {
    Console.WriteLine("C.G");
  }
}
```

the class B provides two override methods: an F method that has the sealed modifier and a G method that does not. B's use of the sealed modifier prevents C from further overriding F. *end example*]

## 15.6.7 Abstract methods

When an instance method declaration includes an abstract modifier, that method is said to be an *abstract method*. Although an abstract method is implicitly also a virtual method, it cannot have the modifier virtual.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, non-abstract derived classes are required to provide their own implementation by overriding that method. Because an abstract method provides no actual implementation, the *method-body* of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes (§15.2.2.2).

[Example: In the following code

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}
public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

the Shape class defines the abstract notion of a geometrical shape object that can paint itself. The Paint method is abstract because there is no meaningful default implementation. The Ellipse and Box classes are concrete Shape implementations. Because these classes are non-abstract, they are required to override the Paint method and provide an actual implementation. *end example*]

It is a compile-time error for a *base-access* (§12.7.9) to reference an abstract method. [*Example*: In the following code

```
abstract class A
{
    public abstract void F();
}
class B: A
{
    public override void F() {
        base.F(); // Error, base.F is abstract
    }
}
```

a compile-time error is reported for the base.F() invocation because it references an abstract method. end example]

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. [*Example*: In the following code

```
using System;
class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}
abstract class B: A
{
    public abstract override void F();
}
class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

class A declares a virtual method, class B overrides this method with an abstract method, and class C overrides the abstract method to provide its own implementation. *end example*]

# 15.6.8 External methods

When a method declaration includes an extern modifier, the method is said to be an *external method*. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method-body* of an external method simply consists of a semicolon. An external method shall not be generic.

The mechanism by which linkage to an external method is achieved, is implementation-defined.

[*Example*: The following example demonstrates the use of the extern modifier and the Dllmport attribute:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;
class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);
    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
```

```
[DllImport("kernel32", SetLastError=true)]
static extern bool SetCurrentDirectory(string name);
}
```

end example]

### 15.6.9 Partial methods

When a method declaration includes a partial modifier, that method is said to be a *partial method*. Partial methods may only be declared as members of partial types (§15.2.7), and are subject to a number of restrictions.

Partial methods may be defined in one part of a type declaration and implemented in another. The implementation is optional; if no part implements the partial method, the partial method declaration and all calls to it are removed from the type declaration resulting from the combination of the parts.

Partial methods shall not define access modifiers; they are implicitly private. Their return type shall be void, and their parameters shall not have the out modifier. The identifier partial is recognized as a contextual keyword (§7.4.4) in a method declaration only if it appears immediately before the void keyword. A partial method cannot explicitly implement interface methods.

There are two kinds of partial method declarations: If the body of the method declaration is a semicolon, the declaration is said to be a *defining partial method declaration*. If the body is given as a *block*, the declaration is said to be an *implementing partial method declaration*. Across the parts of a type declaration, there may be only one defining partial method declaration with a given signature, and there may be only one implementing partial method declaration with a given signature. If an implementing partial method declaration shall exist, and the declarations shall match as specified in the following:

- The declarations shall have the same modifiers (although not necessarily in the same order), method name, number of type parameters and number of parameters.
- Corresponding parameters in the declarations shall have the same modifiers (although not necessarily in the same order) and the same types (modulo differences in type parameter names).
- Corresponding type parameters in the declarations shall have the same constraints (modulo differences in type parameter names).

An implementing partial method declaration can appear in the same part as the corresponding defining partial method declaration.

Only a defining partial method participates in overload resolution. Thus, whether or not an implementing declaration is given, invocation expressions may resolve to invocations of the partial method. Because a partial method always returns void, such invocation expressions will always be expression statements. Furthermore, because a partial method is implicitly private, such statements will always occur within one of the parts of the type declaration within which the partial method is declared.

If no part of a partial type declaration contains an implementing declaration for a given partial method, any expression statement invoking it is simply removed from the combined type declaration. Thus the invocation expression, including any subexpressions, has no effect at run-time. The partial method itself is also removed and will not be a member of the combined type declaration.

If an implementing declaration exists for a given partial method, the invocations of the partial methods are retained. The partial method gives rise to a method declaration similar to the implementing partial method declaration except for the following:

- The partial modifier is not included
- The attributes in the resulting method declaration are the combined attributes of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.

• The attributes on the parameters of the resulting method declaration are the combined attributes of the corresponding parameters of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.

If a defining declaration but not an implementing declaration is given for a partial method M, the following restrictions apply:

- It is a compile-time error to create a delegate from M (§12.7.11.6).
- It is a compile-time error to refer to M inside an anonymous function that is converted to an expression tree type (§9.6).
- Expressions occurring as part of an invocation of M do not affect the definite assignment state (§10.4), which can potentially lead to compile-time errors.
- M cannot be the entry point for an application (§8.1).

Partial methods are useful for allowing one part of a type declaration to customize the behavior of another part, e.g., one that is generated by a tool. Consider the following partial class declaration:

```
partial class Customer
{
   string name;
   public string Name {
     get { return name; }
     set {
        OnNameChanging(value);
        name = value;
        OnNameChanged();
     }
   partial void OnNameChanging(string newName);
   partial void OnNameChanged();
}
```

If this class is compiled without any other parts, the defining partial method declarations and their invocations will be removed, and the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
   string name;
   public string Name {
      get { return name; }
      set { name = value; }
   }
}
```

Assume that another part is given, however, which provides implementing declarations of the partial methods:

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }
    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Then the resulting combined class declaration will be equivalent to the following:

```
class Customer
   string name;
   public string Name {
      get { return name; }
      set {
         OnNameChanging(value);
         name = value;
         OnNameChanged();
      }
   }
   void OnNameChanging(string newName)
      Console.WriteLine("Changing " + name + " to " + newName);
   }
   void OnNameChanged()
   {
      Console.WriteLine("Changed to " + name);
   }
}
```

### 15.6.10 Extension methods

When the first parameter of a method includes the this modifier, that method is said to be an *extension method*. Extension methods shall only be declared in non-generic, non-nested static classes. The first parameter of an extension method may have no modifiers other than this, and the parameter type may not be a pointer type.

[Example: The following is an example of a static class that declares two extension methods:

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }
    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}</pre>
```

#### end example]

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method may be invoked using instance method invocation syntax (§12.7.6.3), using the receiver expression as the first argument.

[Example: The following program uses the extension methods declared above:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

The Slice method is available on the string[], and the ToInt32 method is available on string, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

end example]

# 15.6.11 Method body

The *method-body* of a method declaration consists of either a *block* or a semicolon.

Abstract and external method declarations do not provide a method implementation, so their method bodies simply consist of a semicolon. For any other method, the method body is a block (§13.3) that contains the statements to execute when that method is invoked.

The *effective return type* of a method is void if the return type is void, or if the method is async and the return type is System. Threading. Tasks. Task. Otherwise, the effective return type of a non-async method is its return type, and the effective return type of an async method with return type System. Threading. Tasks. Task.

When the effective return type of a method is void, return statements (§13.10.5) in that method's body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its caller.

When the effective return type of a method is not void, each return statement in that method's body shall specify an expression that is implicitly convertible to the effective return type. The endpoint of the method body of a value-returning method shall not be reachable. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.

[Example: In the following code

```
class A
{
   public int F() {}
                             // Error, return value required
   public int G() {
      return 1;
   }
   public int H(bool b) {
      if (b) {
         return 1;
      }
      else {
         return 0;
      }
   }
}
```

the value-returning F method results in a compile-time error because control can flow off the end of the method body. The G and H methods are correct because all possible execution paths end in a return statement that specifies a return value. *end example*]

# **15.7 Properties**

## 15.7.1 General

A *property* is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have *accessors* that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's characteristics; furthermore, they permit such characteristics to be computed.

Properties are declared using property-declarations:

```
property-declaration:
    attributes<sub>opt</sub> property-modifiers<sub>opt</sub> type member-name { accessor-declarations }
property-modifiers:
    property-modifier
    property-modifiers property-modifier
```

property-modifier:

new public protected internal private static virtual sealed override abstract extern

A property-declaration may include a set of attributes (§22) and a valid combination of the four access modifiers (§15.3.6), the new (§15.3.5), static (§15.7.2), virtual (§15.6.4, §15.7.6), override (§15.6.5, §15.7.6), sealed (§15.6.6), abstract (§15.6.7, §15.7.6), and extern (§15.6.8) modifiers.

Property declarations are subject to the same rules as method declarations (§15.6) with regard to valid combinations of modifiers.

The *type* of a property declaration specifies the type of the property introduced by the declaration, and the *member-name* (§15.6.1) specifies the name of the property. Unless the property is an explicit interface member implementation, the *member-name* is simply an *identifier*. For an explicit interface member implementation (§18.6.2), the *member-name* consists of an *interface-type* followed by a "." and an *identifier*.

The *type* of a property shall be at least as accessible as the property itself (§8.5.5).

The *accessor-declarations*, which shall be enclosed in "{" and "}" tokens, declare the accessors (§15.7.3) of the property. The accessors specify the executable statements associated with reading and writing the property.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a ref or out argument.

When a property declaration includes an extern modifier, the property is said to be an *external property*. Because an external property declaration provides no actual implementation, each of its *accessor-declarations* consists of a semicolon.

## 15.7.2 Static and instance properties

When a property declaration includes a static modifier, the property is said to be a *static property*. When no static modifier is present, the property is said to be an *instance property*.

A static property is not associated with a specific instance, and it is a compile-time error to refer to this in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as this (§12.7.8) in the accessors of that property.

The differences between static and instance members are discussed further in §15.3.8.

### 15.7.3 Accessors

The *accessor-declarations* of a property specify the executable statements associated with reading and writing that property.

```
accessor-declarations:
    get-accessor-declaration set-accessor-declaration<sub>opt</sub>
    set-accessor-declaration get-accessor-declaration<sub>opt</sub>
get-accessor-declaration:
    attributes<sub>opt</sub> accessor-modifier<sub>opt</sub> get accessor-body
set-accessor-declaration:
    attributes<sub>opt</sub> accessor-modifier<sub>opt</sub> set accessor-body
accessor-modifier:
    protected
    internal
    private
    protected internal
    internal protected
accessor-body:
    block
    ;
```

The accessor declarations consist of a *get-accessor-declaration*, a *set-accessor-declaration*, or both. Each accessor declaration consists of optional attributes, an optional *accessor-modifier*, the token get or set, followed by an *accessor-body*.

The use of *accessor-modifiers* is governed by the following restrictions:

- An *accessor-modifier* shall not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no override modifier, an *accessor-modifier* is permitted only if the property or indexer has both a get and set accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an override modifier, an accessor shall match the *accessor*-*modifier*, if any, of the accessor being overridden.
- The *accessor-modifier* shall declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
  - If the property or indexer has a declared accessibility of public, the *accessor-modifier* may be either protected internal, internal, protected, or private.
  - If the property or indexer has a declared accessibility of protected internal, the *accessormodifier* may be either internal, protected, or private.
  - If the property or indexer has a declared accessibility of internal or protected, the *accessormodifier* shall be private.

 If the property or indexer has a declared accessibility of private, no accessor-modifier may be used.

For abstract and extern properties, the *accessor-body* for each accessor specified is simply a semicolon. A non-abstract, non-extern property may be an *automatically implemented property*, in which case both get and set accessors shall be given, both with a semicolon body (§15.7.4). For the accessors of any other non-abstract, non-extern property, the *accessor-body* is a *block* that specifies the statements to be executed when the corresponding accessor is invoked.

A get accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the get accessor of the property is invoked to compute the value of the property (§12.2.2). The body of a get accessor shall conform to the rules for value-returning methods described in §15.6.11. In particular, all return statements in the body of a get accessor shall specify an expression that is implicitly convertible to the property type. Furthermore, the endpoint of a get accessor shall not be reachable.

A set accessor corresponds to a method with a single value parameter of the property type and a void return type. The implicit parameter of a set accessor is always named value. When a property is referenced as the target of an assignment (§12.18), or as the operand of ++ or -- (§12.7.10, 12.8.6), the set accessor is invoked with an argument that provides the new value (§12.18.2). The body of a set accessor shall conform to the rules for void methods described in §15.6.11. In particular, return statements in the set accessor body are not permitted to specify an expression. Since a set accessor implicitly has a parameter named value, it is a compile-time error for a local variable or constant declaration in a set accessor to have that name.

Based on the presence or absence of the get and set accessors, a property is classified as follows:

- A property that includes both a get accessor and a set accessor is said to be a *read-write* property.
- A property that has only a get accessor is said to be a *read-only* property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a set accessor is said to be a *write-only* property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression. [*Note*: The pre- and postfix ++ and -- operators and compound assignment operators cannot be applied to write-only properties, since these operators read the old value of their operand before they write the new one. *end note*]

[Example: In the following code

```
public class Button: Control
   private string caption;
   public string Caption {
      get {
         return caption;
      }
      set
            (caption != value) {
            caption = value;
            Repaint():
         }
      }
   }
   public override void Paint(Graphics g, Rectangle r) {
      // Painting code goes here
   }
}
```

the Button control declares a public Caption property. The get accessor of the Caption property returns the string stored in the private caption field. The set accessor checks if the new value is different

from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The get accessor simply returns a value stored in a private field, and the set accessor modifies that private field and then performs any additional actions required to update fully the state of the object.

Given the Button class above, the following is an example of use of the Caption property:

```
Button okButton = new Button();
okButton.Caption = "OK"; // Invokes set accessor
string s = okButton.Caption; // Invokes get accessor
```

Here, the set accessor is invoked by assigning a value to the property, and the get accessor is invoked by referencing the property in an expression. *end example*]

The get and set accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. [*Example*: The example

```
class A
{
    private string name;
    public string Name {
        get { return name; }
    }
    public string Name {
        set { name = value; }
    }
}
```

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur. *end example*]

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. [*Example*: In the following code

```
class A
{
    public int P {
        set {...}
    }
}
class B: A
{
    new public int P {
        get {...}
}
}
```

the P property in B hides the P property in A with respect to both reading and writing. Thus, in the statements

B b = new B(); b.P = 1; // Error, B.P is read-only ((A)b).P = 1; // Ok, reference to A.P

the assignment to b. P causes a compile-time error to be reported, since the read-only P property in B hides the write-only P property in A. Note, however, that a cast can be used to access the hidden P property. *end example*]

Unlike public fields, properties provide a separation between an object's internal state and its public interface. [*Example*: Consider the following code, which uses a Point struct to represent a location:

```
class Label
{
    private int x, y;
    private string caption;
```

```
public Label(int x, int y, string caption) {
      this.x = x;
      this.y = y;
      this.caption = caption;
   }
   public int X {
      get { return x; }
   public int Y {
   get { return y; }
   }
   public Point Location {
      get { return new Point(x, y); }
   }
   public string Caption {
      get { return caption; }
   }
}
```

Here, the Label class uses two int fields, x and y, to store its location. The location is publicly exposed both as an X and a Y property and as a Location property of type Point. If, in a future version of Label, it becomes more convenient to store the location as a Point internally, the change can be made without affecting the public interface of the class:

```
class Label
   private Point location;
   private string caption;
   public Label(int x, int y, string caption) {
      this.location = new Point(x, y);
      this.caption = caption;
   }
   public int X {
      get { return location.x; }
   }
   public int Y {
      get { return location.y; }
   }
   public Point Location {
      get { return location; }
   }
   public string Caption {
      get { return caption; }
   }
3
```

Had x and y instead been public readonly fields, it would have been impossible to make such a change to the Label class. *end example*]

[*Note*: Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment might replace calls to accessors with the actual code of the accessors. This process is known as *inlining*, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties. *end note*]

[*Example*: Since invoking a get accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for get accessors to have observable side-effects. In the example

```
class Counter
{
private int next;
```

```
public int Next {
    get { return next++; }
}
```

the value of the Next property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side effect, and the property should be implemented as a method instead.

The "no side-effects" convention for get accessors doesn't mean that get accessors should always be written simply to return values stored in fields. Indeed, get accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed get accessor performs no actions that cause observable changes in the state of the object. *end example*]

Properties can be used to delay initialization of a resource until the moment it is first referenced. [Example:

```
using System.IO;
public class Console
   private static TextReader reader;
   private static TextWriter writer;
   private static TextWriter error;
   public static TextReader In {
      get
         {
if
            (reader == null) {
            reader = new StreamReader(Console.OpenStandardInput());
         return reader;
      }
   }
   public static TextWriter Out {
      get
            (writer == null) {
            writer = new StreamWriter(Console.OpenStandardOutput());
         return writer;
      }
   }
   public static TextWriter Error {
         {
if
      get
            (error == null) {
            error = new StreamWriter(Console.OpenStandardError());
         }
         return error;
      }
   }
}
```

The Console class contains three properties, In, Out, and Error, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the Console class can delay their initialization until they are actually used. For example, upon first referencing the Out property, as in

Console.Out.WriteLine("hello, world");

the underlying TextWriter for the output device is created. However, if the application makes no reference to the In and Error properties, then no objects are created for those devices. *end example*]

# 15.7.4 Automatically implemented properties

When a property is specified as an automatically implemented property, a hidden backing field is automatically available for the property, and the accessors are implemented to read from and write to that

backing field. The hidden backing field is inaccessible, it can be read and written only through the automatically implemented property accessors, even within the containing type.

[Example:

```
public class Point {
    public int X { get; set; } // automatically implemented
    public int Y { get; set; } // automatically implemented
}
```

is equivalent to the following declaration:

```
public class Point {
    private int x;
    private int y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

end example]

Because the backing field is inaccessible, automatically implemented read-only or write-only properties do not make sense, and are disallowed. It is however possible to set the access level of each accessor differently. Thus, the effect of a read-only property with a private backing field can be mimicked like this:

```
public class ReadOnlyPoint {
    public int X { get; private set; }
    public int Y { get; private set; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

## 15.7.5 Accessibility

If an accessor has an *accessor-modifier*, the accessibility domain (§8.5.3) of the accessor is determined using the declared accessibility of the *accessor-modifier*. If an accessor does not have an *accessor-modifier*, the accessibility domain of the accessor is determined from the declared accessibility of the property or indexer.

The presence of an *accessor-modifier* never affects member lookup (§12.5) or overload resolution (§12.6.4). The modifiers on the property or indexer always determine which property or indexer is bound to, regardless of the context of the access.

Once a particular property or indexer has been selected, the accessibility domains of the specific accessors involved are used to determine if that usage is valid:

- If the usage is as a value (§12.2.2), the get accessor shall exist and be accessible.
- If the usage is as the target of a simple assignment (§12.18.2), the set accessor shall exist and be accessible.
- If the usage is as the target of compound assignment (§12.18.3), or as the target of the ++ or -operators (§12.7.10, §12.8.6), both the get accessors and the set accessor shall exist and be
  accessible.

[*Example*: In the following example, the property A.Text is hidden by the property B.Text, even in contexts where only the set accessor is called. In contrast, the property B.Count is not accessible to class M, so the accessible property A.Count is used instead.

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }
}
```

```
public int Count {
       get { return 5; }
set { }
   }
}
class B: A
   private string text = "goodbye";
   private int count = 0;
   new public string Text {
       get { return text; }
       protected set { text = value; }
   }
   new protected int Count {
       get { return count; }
set { count = value; }
   }
}
class M
   static void Main() {
       B b = new B();
                                 // Calls A.Count set accessor
       b.Count = 12;
       int i = b.Count;
b.Text = "howdy";
                                // Calls A.Count get accessor
// Error, B.Text set accessor not accessible
                                // Calls B.Text get accessor
       string s = b.Text;
   }
}
```

### end example]

An accessor that is used to implement an interface shall not have an *accessor-modifier*. If only one accessor is used to implement an interface, the other accessor may be declared with an *accessor-modifier*: [*Example*:

```
public interface I
{
    string Prop { get; }
}
public class C: I
{
    public Prop {
        get { return "April"; } // Must not have a modifier here
        internal set {...} // Ok, because I.Prop has no set accessor
    }
}
```

end example]

### 15.7.6 Virtual, sealed, override, and abstract accessors

A virtual property declaration specifies that the accessors of the property are virtual. The virtual modifier applies to all non-private accessors of a property. When an accessor of a virtual property has the private accessor is implicitly not virtual.

An abstract property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an abstract property declaration provides no actual implementation, its *accessor-body* simply consists of a semicolon. An abstract property shall not have a private accessor.

A property declaration that includes both the abstract and override modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are only permitted in abstract classes (§15.2.2.2). The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an override directive. This is known as an **overriding property declaration**. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion between the type of the overriding and the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property can include either a single accessor or both accessors.

An overriding property declaration may include the sealed modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in §15.6.4, §15.6.5, §15.6.6, and §15.6.7 apply as if accessors were methods of a corresponding form:

- A get accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
- A set accessor corresponds to a method with a single value parameter of the property type, a void return type, and the same modifiers as the containing property.

[Example: In the following code

```
abstract class A
{
    int y;
    public virtual int X {
        get { return 0; }
    }
    public virtual int Y {
        get { return y; }
        set { y = value; }
    }
    public abstract int Z { get; set; }
}
```

X is a virtual read-only property, Y is a virtual read-write property, and Z is an abstract read-write property. Because Z is abstract, the containing class A shall also be declared abstract.

A class that derives from A is show below:

```
class B: A
{
    int z;
    public override int x {
        get { return base.x + 1; }
    }
    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }
    public override int z {
        get { return z; }
        set { z = value; }
    }
}</pre>
```

Here, the declarations of X, Y, and Z are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The get accessor of X and the set accessor of Y use the base keyword to access the inherited accessors. The declaration of Z overrides both abstract accessors—thus, there are no outstanding abstract function members in B, and B is permitted to be a non-abstract class. *end example*]

When a property is declared as an override, any overridden accessors shall be accessible to the overriding code. In addition, the declared accessibility of both the property or indexer itself, and of the accessors, shall match that of the overridden member and accessors. [*Example*:

```
public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}
public class D: B
{
    public override int P {
        protected set {...}
        // Must specify protected here
        get {...}
    }
}
```

end example]

# 15.8 Events

### 15.8.1 General

An *event* is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying *event handlers*.

Events are declared using event-declarations:

```
event-declaration:
    attributes<sub>opt</sub> event-modifiers<sub>opt</sub> event type variable-declarators ;
    attributes<sub>opt</sub> event-modifiers<sub>opt</sub> event type member-name
            { event-accessor-declarations }
event-modifiers:
    event-modifier
    event-modifiers event-modifier
event-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern
event-accessor-declarations:
    add-accessor-declaration remove-accessor-declaration
    remove-accessor-declaration add-accessor-declaration
```

add-accessor-declaration: attributes<sub>opt</sub> add block remove-accessor-declaration:

attributes<sub>opt</sub> remove block

An *event-declaration* may include a set of *attributes* (§22) and a valid combination of the four access modifiers (§15.3.6), the new (§15.3.5), static (§15.6.3, §15.8.4), virtual (§15.6.4, §15.8.5), override (§15.6.5, §15.8.5), sealed (§15.6.6), abstract (§15.6.7, §15.8.5), and extern (§15.6.8) modifiers.

Event declarations are subject to the same rules as method declarations (§15.6) with regard to valid combinations of modifiers.

The *type* of an event declaration shall be a *delegate-type* (§9.2.8), and that *delegate-type* shall be at least as accessible as the event itself (§8.5.5).

An event declaration can include *event-accessor-declarations*. However, if it does not, for non-extern, non-abstract events, the compiler shall supply them automatically (§15.8.2); for extern events, the accessors are provided externally.

An event declaration that omits *event-accessor-declarations* defines one or more events—one for each of the *variable-declarators*. The attributes and modifiers apply to all of the members declared by such an *event-declaration*.

It is a compile-time error for an *event-declaration* to include both the abstract modifier and *event-accessor-declarations*.

When an event declaration includes an extern modifier, the event is said to be an *external event*. Because an external event declaration provides no actual implementation, it is an error for it to include both the extern modifier and *event-accessor-declarations*.

It is a compile-time error for a *variable-declarator* of an event declaration with an abstract or external modifier to include a *variable-initializer*.

An event can be used as the left-hand operand of the += and -= operators. These operators are used, respectively, to attach event handlers to, or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

The only operations that are permitted on an event by code that is outside the type in which that event is declared, are += and -=. Therefore, while such code can add and remove handlers for an event, it cannot directly obtain or modify the underlying list of event handlers.

In an operation of the form x += y or x -= y, when x is an event the result of the operation has type void (§12.18.4) (as opposed to having the type of x, with the value of x after the assignment, as for other the += and -= operators defined on non-event types). This prevents external code from indirectly examining the underlying delegate of an event.

[Example: The following example shows how event handlers are attached to instances of the Button class:

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
}
public class LoginDialog: Form
{
    Button okButton;
    Button cancelButton;
```

```
public LoginDialog() {
    okButton = new Button(...);
    okButton.Click += new EventHandler(OkButtonClick);
    cancelButton = new Button(...);
    cancelButton.Click += new EventHandler(CancelButtonClick);
}
void OkButtonClick(object sender, EventArgs e) {
    // Handle okButton.Click event
}
void CancelButtonClick(object sender, EventArgs e) {
    // Handle cancelButton.Click event
}
```

Here, the LoginDialog instance constructor creates two Button instances and attaches event handlers to the Click events. *end example*]

## 15.8.2 Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event shall not be abstract or extern, and shall not explicitly include *event-accessor-declarations*. Such an event can be used in any context that permits a field. The field contains a delegate (§20), which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains null.

[*Example*: In the following code

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
    protected void OnClick(EventArgs e) {
        EventHandler handler = Click;
        if (handler != null)
            handler(this, e);
    }
    public void Reset() {
        Click = null;
     }
}
```

Click is used as a field within the Button class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The OnClick method in the Button class "raises" the Click event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null and that the check is made on a local copy to ensure thread safety.

Outside the declaration of the Button class, the Click member can only be used on the left-hand side of the += and -= operators, as in

b.Click += new EventHandler(...);

which appends a delegate to the invocation list of the Click event, and

b.Click -= new EventHandler(...);

which removes a delegate from the invocation list of the Click event. end example]

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock

(§10.4.4.19) on the containing object for an instance event, or the type object (§12.7.11.7) for a static event.

[Note: Thus, an instance event declaration of the form:

```
class X
{
    public event D Ev;
}
```

shall be compiled to something equivalent to:

```
class X
{
   private D __Ev; // field to hold the delegate
   public event D Ev {
      add {
          /* add the delegate in a thread safe way */
      }
      remove {
          /* remove the delegate in a thread safe way */
      }
}
```

Within the class X, references to Ev on the left-hand side of the += and -= operators cause the add and remove accessors to be invoked. All other references to Ev are compiled to reference the hidden field \_\_\_Ev instead (\$12.7.5). The name "\_\_\_Ev" is arbitrary; the hidden field could have any name or no name at all. *end note*]

## 15.8.3 Event accessors

[Note: Event declarations typically omit *event-accessor-declarations*, as in the Button example above. For example, they might be included if the storage cost of one field per event is not acceptable. In such cases, a class can include *event-accessor-declarations* and use a private mechanism for storing the list of event handlers. *end note*]

The *event-accessor-declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add-accessor-declaration* and a *remove-accessor-declaration*. Each accessor declaration consists of the token add or remove followed by a *block*. The *block* associated with an *add-accessor-declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove-accessor-declaration* specifies the statements to execute when an event handler is removed.

Each *add-accessor-declaration* and *remove-accessor-declaration* corresponds to a method with a single value parameter of the event type, and a void return type. The implicit parameter of an event accessor is named value. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is += then the add accessor is used, and if the assignment operator is -= then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add-accessor-declaration* or a *remove-accessor-declaration* shall conform to the rules for void methods described in §15.6.9. In particular, return statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named value, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

[Example: In the following code

```
class Control: Component
   // Unique keys for events
static readonly object mouseDownEventKey = new object();
static readonly object mouseUpEventKey = new object();
   // Return event handler associated with key
   protected Delegate GetEventHandler(object key) {...}
   // Add event handler associated with key
   protected void AddEventHandler(object key, Delegate handler) {...}
   // Remove event handler associated with key
   protected void RemoveEventHandler(object key, Delegate handler) {...}
   // MouseDown event
   public event MouseEventHandler MouseDown {
       add { AddEventHandler(mouseDownEventKey, value); }
       remove { RemoveEventHandler(mouseDownEventKey, value); }
   }
   // MouseUp event
   public event MouseEventHandler MouseUp {
       add { AddEventHandler(mouseUpEventKey, value); }
       remove { RemoveEventHandler(mouseUpEventKey, value); }
   }
   // Invoke the MouseUp event
   protected void OnMouseUp(MouseEventArgs args) {
      MouseEventHandler handler;
handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
       if (handler != null)
          handler(this, args);
   }
}
```

the Control class implements an internal storage mechanism for events. The AddEventHandler method associates a delegate value with a key, the GetEventHandler method returns the delegate currently associated with a key, and the RemoveEventHandler method removes a delegate as an event handler for the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for associating a null delegate value with a key, and thus unhandled events consume no storage. *end example*]

### 15.8.4 Static and instance events

When an event declaration includes a static modifier, the event is said to be a *static event*. When no static modifier is present, the event is said to be an *instance event*.

A static event is not associated with a specific instance, and it is a compile-time error to refer to this in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as this (§12.7.8) in the accessors of that event.

The differences between static and instance members are discussed further in §15.3.8.

### 15.8.5 Virtual, sealed, override, and abstract accessors

A virtual event declaration specifies that the accessors of that event are virtual. The virtual modifier applies to both accessors of an event.

An abstract event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an accessor for an abstract event declaration provides no actual implementation, it shall not provide *event-accessor-declarations*.

An event declaration that includes both the abstract and override modifiers specifies that the event is abstract and overrides a base event. The accessors of such an event are also abstract.

Abstract event declarations are only permitted in abstract classes (§15.2.2.2).

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an override modifier. This is known as an **overriding event declaration**. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration shall specify the exact same accessibility modifiers and name as the overridden event, there shall be an identity conversion between the type of the overriding and the overridden event, and both the add and remove accessors shall be specified within the declaration.

An overriding event declaration can include the sealed modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed.

It is a compile-time error for an overriding event declaration to include a new modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in §15.6.4, §15.6.5, §15.6.6, and §15.6.7 apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a void return type, and the same modifiers as the containing event.

### 15.9 Indexers

An *indexer* is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer-declarations*:

```
indexer-declaration:
   attributes<sub>opt</sub> indexer-modifiers<sub>opt</sub> indexer-declarator { accessor-declarations }
indexer-modifiers:
   indexer-modifier
   indexer-modifiers indexer-modifier
indexer-modifier:
   new
   public
   protected
    internal
   private
   virtual
   sealed
   override
   abstract
   extern
indexer-declarator:
   type this [ formal-parameter-list ]
   type interface-type . this [ formal-parameter-list ]
```

An *indexer-declaration* may include a set of *attributes* (§22) and a valid combination of the four access modifiers (§15.3.6), the new (§15.3.5), virtual (§15.6.4), override (§15.6.5), sealed (§15.6.6), abstract (§15.6.7), and extern (§15.6.8) modifiers.

Indexer declarations are subject to the same rules as method declarations (§15.6) with regard to valid combinations of modifiers, with the one exception being that the static modifier is not permitted on an indexer declaration.

The modifiers virtual, override, and abstract are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract indexer can override a virtual one.

The *type* of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the *type* is followed by the keyword this. For an explicit interface member implementation, the *type* is followed by an *interface-type*, a ".", and the keyword this. Unlike other members, indexers do not have user-defined names.

The *formal-parameter-list* specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method (§15.6.2), except that at least one parameter shall be specified, and that the this, ref, and out parameter modifiers are not permitted.

The *type* of an indexer and each of the types referenced in the *formal-parameter-list* shall be at least as accessible as the indexer itself (§8.5.5).

The *accessor-declarations* (§15.7.3), which shall be enclosed in "{" and "}" tokens, declare the accessors of the indexer. The accessors specify the executable statements associated with reading and writing indexer elements.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer element is not classified as a variable. Thus, it is not possible to pass an indexer element as a ref or out argument.

The *formal-parameter-list* of an indexer defines the signature (§8.6) of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer's signature.

The signature of an indexer shall differ from the signatures of all other indexers declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

- A property is identified by its name, whereas an indexer is identified by its signature.
- A property is accessed through a *simple-name* (§12.7.3) or a *member-access* (§12.7.5), whereas an indexer element is accessed through an *element-access* (§12.7.7.3).
- A property can be a static member, whereas an indexer is always an instance member.
- A get accessor of a property corresponds to a method with no parameters, whereas a get accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A set accessor of a property corresponds to a method with a single parameter named value, whereas a set accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named value.
- It is a compile-time error for an indexer accessor to declare a local variable or local constant with the same name as an indexer parameter.
- In an overriding property declaration, the inherited property is accessed using the syntax base.P, where P is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax base[E], where E is a comma-separated list of expressions.

Aside from these differences, all rules defined in §15.7.3 and §15.7.6 apply to indexer accessors as well as to property accessors.

When an indexer declaration includes an extern modifier, the indexer is said to be an *external indexer*. Because an external indexer declaration provides no actual implementation, each of its *accessor-declarations* consists of a semicolon.

[*Example*: The example below declares a BitArray class that implements an indexer for accessing the individual bits in the bit array.

```
using System;
class BitArray
{
int[] bits;
int length;
```

```
public BitArray(int length) {
      if (length < 0) throw new ArgumentException();
      bits = new int[((length - 1) >> 5) + 1];
this.length = length;
   }
   public int Length {
      get { return length; }
   public bool this[int index] {
          :{
if
      get
             (index < 0 || index >= length) 
             throw new IndexOutOfRangeException();
          return (bits[index >> 5] & 1 << index) != 0;
      }
      set {
if
             (index < 0 || index >= length) 
             throw new IndexOutOfRangeException();
          if
             (value) {
             bits[index >> 5] |= 1 << index;</pre>
          }
          else {
             bits[index >> 5] &= ~(1 << index);</pre>
          }
      }
   }
}
```

An instance of the BitArray class consumes substantially less memory than a corresponding bool[] (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same operations as a bool[].

The following CountPrimes class uses a BitArray and the classical "sieve" algorithm to compute the number of primes between2 and a given maximum:

```
class CountPrimes
   static int Count(int max) {
     BitArray flags = new BitArray(max + 1);
     for (int j´=`i * 2; j <= max; j += i) flags[j] = true;</pre>
           count++;
        }
     ł
     return count;
  }
  static void Main(string[] args) {
      int max = int.Parse(args[0]);
     int count = Count(max);
     Console.WriteLine(
         'Found {0} primes between 2 and {1}", count, max);
  }
}
```

Note that the syntax for accessing elements of the BitArray is precisely the same as for a bool[]. end example]

[*Example*: The following example shows a  $26 \times 10$  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.

using System;

```
class Grid
ł
   const int NumRows = 26;
   const int NumCols = 10;
int[,] cells = new int[NumRows, NumCols];
   public int this[char row, int col]
       get {
          row = Char.ToUpper(row);
if (row < 'A' || row > 'Z') {
              throw new ArgumentOutOfRangeException("row");
          if
             (col < 0 || col >= NumCols) {
              throw new ArgumentOutOfRangeException ("col");
          }
          return cells[row - 'A', col];
       }
       set {
          row = Char.ToUpper(row);
if (row < 'A' || row > 'Z') {
throw new ArgumentOutOfRangeException ("row");
          throw new ArgumentOutOfRangeException ("col");
          cells[row - 'A', col] = value;
       }
   }
}
```

```
end example]
```

## 15.10 Operators

### 15.10.1 General

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**:

```
operator-declaration:
   attributes<sub>opt</sub> operator-modifiers operator-declarator operator-body
operator-modifiers:
   operator-modifier
   operator-modifiers operator-modifier
operator-modifier:
   public
   static
   extern
operator-declarator:
   unary-operator-declarator
   binary-operator-declarator
   conversion-operator-declarator
unary-operator-declarator:
   type operator overloadable-unary-operator (fixed-parameter)
overloadable-unary-operator: one of
   +
        _
             1
                  ~
                       ++
                              ___
                                    true
                                             false
binary-operator-declarator:
   type operator overloadable-binary-operator (fixed-parameter, fixed-parameter)
```

```
overloadable-binary-operator: one of
                                %
                                        &
                                               ٨
                                                                     right-shift
   +
           _
                                                              <<
                         /
           !=
                  >
                         <
   ==
                                >=
                                        <=
conversion-operator-declarator:
   implicit operator type ( fixed-parameter )
   explicit operator type ( fixed-parameter )
operator-body:
   block
   ;
```

There are three categories of overloadable operators: Unary operators (§15.10.2), binary operators (§15.10.3), and conversion operators (§15.10.4).

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 shall conform to the rules for value-returning methods described in §15.6.11.

The following rules apply to all operator declarations:

- An operator declaration shall include both a public and a static modifier.
- The parameter(s) of an operator shall have no modifiers.
- The signature of an operator (§15.10.2, §15.10.3, §15.10.4) shall differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration shall be at least as accessible as the operator itself (§8.5.5).
- 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 subclauses.

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 §12.4.

Additional information on conversion operators can be found in §11.5.

#### 15.10.2 Unary operators

The following rules apply to unary operator declarations, where T denotes the instance type of the class or struct that contains the operator declaration:

- A unary +, -, !, or ~ operator shall take a single parameter of type T or T? and can return any type.
- A unary ++ or -- operator shall take a single parameter of type T or T? and shall return that same type or a type derived from it.
- A unary true or false operator shall take a single parameter of type T or T? and shall 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 §12.21.

[*Example*: The following example shows an implementation and subsequent usage of operator++ for an integer vector class:

```
public class IntVector
    public IntVector(int length) {...}
    public int Length { ... } // read-only property
public int this[int index] { ... } // read-write indexer
    public static IntVector operator++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
for (int i = 0; i < iv.Length; i++)</pre>
            temp[i] = iv[i] + 1;
        return temp;
    }
}
class Test
ł
    static void Main() {
        IntVector iv1 = new IntVector(4); // vector of 4 x 0
        IntVector iv2;
                            // iv2 contains 4 x 0, iv1 contains 4 x 1
// iv2 contains 4 x 2, iv1 contains 4 x 2
        iv2 = iv1++;
        iv2 = ++iv1;
    }
}
```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (§12.7.10), and the prefix increment and decrement operators (§12.8.6). Unlike in C++, this method should not modify the value of its operand directly as this would violate the standard semantics of the postfix increment operator (§12.8.6). *end example*]

# 15.10.3 Binary operators

The following rules apply to binary operator declarations, where T denotes the instance type of the class or struct that contains the operator declaration:

- A binary non-shift operator shall take two parameters, at least one of which shall have type T or T?, and can return any type.
- A binary << or >> operator (§12.10) shall take two parameters, the first of which shall have type T or T? and the second of which shall have type int or int?, and can return any type.

The signature of a binary operator consists of the operator token  $(+, -, *, /, \%, \&, |, \land, <<, >>,, ==, !=, >, <, >=, or <=)$  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 shall be a matching declaration of the other operator of the pair. Two operator declarations match if identity conversions exist between their return types and their corresponding parameter types. The following operators require pair-wise declaration:

- operator == and operator !=
- operator > and operator <</li>
- operator >= and operator <=</li>

### 15.10.4 Conversion operators

A conversion operator declaration introduces a *user-defined conversion* (§11.5), which augments the predefined implicit and explicit conversions. A conversion operator declaration that includes the implicit keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in §11.2.

A conversion operator declaration that includes the explicit keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions, and are described further in §11.3.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator.

For a given source type S and target type T, if S or T are nullable value types, let  $S_0$  and  $T_0$  refer to their underlying types; otherwise,  $S_0$  and  $T_0$  are equal to S and T respectively. A class or struct is permitted to declare a conversion from a source type S to a target type T only if all of the following are true:

- S<sub>0</sub> and T<sub>0</sub> are different types.
- Either S<sub>0</sub> or T<sub>0</sub> is the instance type of the class or struct that contains the operator declaration.
- Neither S<sub>0</sub> nor T<sub>0</sub> is an *interface-type*.
- Excluding user-defined conversions, a conversion does not exist from S to T or from T to S.

For the purposes of these rules, any type parameters associated with S or T are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

[*Example*: In the following:

```
class C<T> {...}
class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...} // Ok
    public static implicit operator C<string>(D<T> value) {...} // Ok
    public static implicit operator C<T>(D<T> value) {...} // Error
}
```

the first two operator declarations are permitted because T and int and string, respectively are considered unique types with no relationship. However, the third operator is an error because C<T> is the base class of D<T>. *end example*]

From the second rule, it follows that a conversion operator shall convert either to or from the class or struct type in which the operator is declared. [*Example*: It is possible for a class or struct type C to define a conversion from C to int and from int to C, but not from int to bool. *end example*]

It is not possible to directly redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to object because implicit and explicit conversions already exist between object and all other types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since a conversion would then already exist. However, it *is* possible to declare operators on generic types that, for particular type arguments, specify conversions that already exist as pre-defined conversions. [*Example*:

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

when type object is specified as a type argument for T, the second operator declares a conversion that already exists (an implicit, and therefore also an explicit, conversion exists from any type to type object). *end example*]

In cases where a pre-defined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

- If a pre-defined implicit conversion (§11.2) exists from type S to type T, all user-defined conversions (implicit or explicit) from S to T are ignored.
- If a pre-defined explicit conversion (§11.3) exists from type S to type T, any user-defined explicit conversions from S to T are ignored. Furthermore:
  - o If either S or T is an interface type, user-defined implicit conversions from S to T are ignored.
  - Otherwise, user-defined implicit conversions from S to T are still considered.

For all types but object, the operators declared by the Convertible<T> type above do not conflict with pre-defined conversions. [*Example*:

However, for type object, pre-defined conversions hide the user-defined conversions in all cases but one:

#### end example]

User-defined conversions are not allowed to convert from or to *interface-types*. In particular, this restriction ensures that no user-defined transformations occur when converting to an *interface-type*, and that a conversion to an *interface-type* succeeds only if the object being converted actually implements the specified *interface-type*.

The signature of a conversion operator consists of the source type and the target type. (This is the only form of member for which the return type participates in the signature.) The implicit or explicit classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an implicit and an explicit conversion operator with the same source and target types.

[*Note*: In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion. *end note*]

[Example: In the following code

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

the conversion from Digit to byte is implicit because it never throws exceptions or loses information, but the conversion from byte to Digit is explicit since Digit can only represent a subset of the possible values of a byte. *end example*]

# 15.11 Instance constructors

### 15.11.1 General

An *instance constructor* is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor-declarations*:

```
constructor-declaration:
    attributes<sub>opt</sub> constructor-modifiers<sub>opt</sub> constructor-declarator constructor-body
constructor-modifiers:
    constructor-modifier
    constructor-modifiers constructor-modifier
constructor-modifier:
    public
    protected
    internal
    private
    extern
constructor-declarator:
    identifier ( formal-parameter-list<sub>opt</sub> ) constructor-initializer<sub>opt</sub>
constructor-initializer:
    : base ( argument-list<sub>opt</sub> )
    : this ( argument-list<sub>opt</sub> )
constructor-body:
    block
    ;
```

A constructor-declaration may include a set of attributes (§22), a valid combination of the four access modifiers (§15.3.6), and an extern (§15.6.8) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The *identifier* of a *constructor-declarator* shall name the class in which the instance constructor is declared. If any other name is specified, a compile-time error occurs.

The optional *formal-parameter-list* of an instance constructor is subject to the same rules as the *formal-parameter-list* of a method (§15.6). As the this modifier for parameters only applies to extension methods (§15.6.10), no parameter in a constructor's *formal-parameter-list* shall contain the this modifier. The formal parameter list defines the signature (§8.6) of an instance constructor and governs the process whereby overload resolution (§12.6.4) selects a particular instance constructor in an invocation.

Each of the types referenced in the *formal-parameter-list* of an instance constructor shall be at least as accessible as the constructor itself (§8.5.5).

The optional *constructor-initializer* specifies another instance constructor to invoke before executing the statements given in the *constructor-body* of this instance constructor. This is described further in §15.11.2.

When a constructor declaration includes an extern modifier, the constructor is said to be an *external constructor*. Because an external constructor declaration provides no actual implementation, its *constructor-body* consists of a semicolon. For all other constructors, the *constructor-body* consists of a *block*, which specifies the statements to initialize a new instance of the class. This corresponds exactly to the *block* of an instance method with a void return type (§15.6.11).

Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually declared in the class, with the exception that if a class contains no instance constructor declarations, a default instance constructor is automatically provided (§15.11.5).

Instance constructors are invoked by *object-creation-expressions* (§12.7.11.2) and through *constructor-initializers*.

# 15.11.2 Constructor initializers

All instance constructors (except those for class object) implicitly include an invocation of another instance constructor immediately before the *constructor-body*. The constructor to implicitly invoke is determined by the *constructor-initializer*:

- An instance constructor initializer of the form base (*argument-list<sub>opt</sub>*) causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument-list* and the overload resolution rules of §12.6.4. The set of candidate instance constructors consists of all the accessible instance constructors of the direct base class. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
- An instance constructor initializer of the form this (*argument-list*<sub>opt</sub>) invokes another instance constructor from the same class. The constructor is selected using *argument-list* and the overload resolution rules of §12.6.4. The set of candidate instance constructors consists of all instance constructors declared in the class itself. If the resulting set of applicable instance constructors is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration invokes itself through a chain of one or more constructor initializers, a compile-time error occurs.

If an instance constructor has no constructor initializer, a constructor initializer of the form base() is implicitly provided. [*Note*: Thus, an instance constructor declaration of the form

C(...) {...}

is exactly equivalent to

C(...): base() {...}

#### end note]

The scope of the parameters given by the *formal-parameter-list* of an instance constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. [*Example*:

```
class A
{
   public A(int x, int y) {}
}
class B: A
{
   public B(int x, int y): base(x + y, x - y) {}
}
```

#### end example]

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference this in an argument expression of the constructor initializer, as it is a compile-time error for an argument expression to reference any instance member through a *simple-name*.

### 15.11.3 Instance variable initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form base(...), that constructor implicitly performs the initializations specified by the *variable-initializers* of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class

constructor. The variable initializers are executed in the textual order in which they appear in the class declaration (§15.5.6).

# 15.11.4 Constructor execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed *before* the invocation of the base class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before *any* statements that have access to that instance are executed. [*Example*: Given the following:

```
using System;
class A
{
   public A() {
     PrintFields();
   public virtual void PrintFields() {}
}
class B: A
   int x = 1;
   int y;
   public B() {
      y = -1;
   }
   public override void PrintFields() {
      Console.WriteLine("x = \{0\}, y = \{1\}", x, y);
   }
}
```

when new B() is used to create an instance of B, the following output is produced:

x = 1, y = 0

The value of x is 1 because the variable initializer is executed before the base class instance constructor is invoked. However, the value of y is 0 (the default value of an int) because the assignment to y is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor-body*. The example

```
using System;
using System.Collections;
class A
{
   int x = 1, y = -1, count;
   public A() {
      count = 0;
   ş
   public A(int n) {
      count = n;
}
class B: A
   double sqrt2 = Math.Sqrt(2.0);
   ArrayList items = new ArrayList(100);
   int max;
   public B(): this(100) {
      items.Add("default");
```

```
public B(int n): base(n - 1) {
    max = n;
}
```

contains several variable initializers; it also contains constructor initializers of both forms (base and this). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```
using System.Collections:
class A
{
    int x, y, count;
   public A() {
       x = 1;
                                               // Variable initializer
                                               // Variable initializer
       y = -1;
       object();
                                               // Invoke object() constructor
        count = 0;
    }
   public A(int n) {
                                               // Variable initializer
// Variable initializer
// Invoke object() constructor
       x = 1;
       y = -1;
       object();
        count = n;
   }
}
class B: A
   double sqrt2;
   ArrayList items;
    int max;
   public B(): this(100) {
       B(100);
                                               // Invoke B(int) constructor
        items.Add("default");
   }
   public B(int n): base(n - 1) {
                                              // Variable initializer
// Variable initializer
// Invoke A(int) constructor
       sqrt2 = Math.Sqrt(2.0);
items = new ArrayList(100);
       A(n - 1);
       max = n;
   }
}
```

end example]

# 15.11.5 Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes a constructor of the direct base class, as if it had a constructor initializer of the form base(). If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. [Note: Thus, the default constructor is always of the form

protected C(): base() {}

or

public C(): base() {}

where C is the name of the class. end note]

If overload resolution is unable to determine a unique best candidate for the base-class constructor initializer then a compile-time error occurs.

[Example: In the following code

```
class Message
{
    object sender;
    string text;
}
```

a default constructor is provided because the class contains no instance constructor declarations. Thus, the example is precisely equivalent to

```
class Message
{
    object sender;
    string text;
    public Message(): base() {}
}
```

end example]

# 15.12 Static constructors

A *static constructor* is a member that implements the actions required to initialize a closed class. Static constructors are declared using *static-constructor-declarations*:

```
static-constructor-declaration:
    attributes<sub>opt</sub> static-constructor-modifiers identifier ( ) static-constructor-body
static-constructor-modifiers:
    extern<sub>opt</sub> static
    static extern<sub>opt</sub>
static-constructor-body:
    block
    ;
A static-constructor-declaration may include a set of attributes (§22) and an extern modifier (§15.6.8).
```

The *identifier* of a *static-constructor-declaration* shall name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

When a static constructor declaration includes an extern modifier, the static constructor is said to be an *external static constructor*. Because an external static constructor declaration provides no actual implementation, its *static-constructor-body* consists of a semicolon. For all other static constructor declarations, the *static-constructor-body* consists of a *block*, 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 (§15.6.11).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a closed 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:

- An instance of the class is created.
- Any of the static members of the class are referenced.

If a class contains the Main method (§8.1) in which execution begins, the static constructor for that class executes before the Main method is called.

To initialize a new closed class type, first a new set of static fields (§15.5.2) for that particular closed type is created. Each of the static fields is initialized to its default value (§15.5.5). Next, the static field initializers (§15.5.6.2) are executed for those static fields. Finally, the static constructor is executed.[*Example*: The example

```
using System;
class Test
{
   static void Main() {
    A.F();
        B.F();
    }
}
class A
    static A() {
        Console.WriteLine("Init A");
    }
   public static void F() {
    Console.WriteLine("A.F");
    }
}
class B
    static B() {
        Console.WriteLine("Init B");
    }
   public static void F() {
    Console.WriteLine("B.F");
    }
}
```

must produce the output:

```
Init A
A.F
Init B
B.F
```

because the execution of A's static constructor is triggered by the call to A.F, and the execution of B's static constructor is triggered by the call to B.F. *end example*]

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

[Example: The example

```
using System;
class A
{
    public static int X;
    static A() {
        X = B.Y + 1;
    }
}
class B
{
    public static int Y = A.X + 1;
    static B() {}
    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

produces the output

X = 1, Y = 2

To execute the Main method, the system first runs the initializer for B.Y, prior to class B's static constructor. Y's initializer causes A's static constructor to be run because the value of A.X is referenced. The static constructor of A in turn proceeds to compute the value of X, and in doing so fetches the default

value of Y, which is zero. A.X is thus initialized to 1. The process of running A's static field initializers and static constructor then completes, returning to the calculation of the initial value of Y, the result of which becomes 2. *end example*]

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints (§15.2.5). [*Example*: The following type uses a static constructor to enforce that the type argument is an enum:

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

end example]

# 15.13 Finalizers

[*Note*: In an earlier version of this standard, what is now referred to as a "finalizer" was called a "destructor". Experience has shown that the term "destructor" caused confusion and often resulted to incorrect expectations, especially to programmers knowing C++. In C++, a destructor is called in a determinate manner, whereas, in C#, a finalizer is not. To get determinate behavior from C#, one should use Dispose. *end note*]

A *finalizer* is a member that implements the actions required to finalize an instance of a class. A finalizer is declared using a *finalizer-declaration*:

```
finalizer-declaration:
    attributes<sub>opt</sub> extern<sub>opt</sub> ~ identifier ( ) finalizer-body
finalizer-body:
    block
;
```

A finalizer-declaration may include a set of attributes (§22).

The *identifier* of a *finalizer-declarator* shall name the class in which the finalizer is declared. If any other name is specified, a compile-time error occurs.

When a finalizer declaration includes an extern modifier, the finalizer is said to be an *external finalizer*. Because an external finalizer declaration provides no actual implementation, its *finalizer-body* consists of a semicolon. For all other finalizers, the *finalizer-body* consists of a *block*, which specifies the statements to execute in order to finalize an instance of the class. A *finalizer-body* corresponds exactly to the *method-body* of an instance method with a void return type (§15.6.11).

Finalizers are not inherited. Thus, a class has no finalizers other than the one that may be declared in that class.

[*Note*: Since a finalizer is required to have no parameters, it cannot be overloaded, so a class can have, at most, one finalizer. *end note*]

Finalizers are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for finalization when it is no longer possible for any code to use that instance. Execution of the finalizer for the instance may occur at any time after the instance becomes eligible for finalization (§8.9). When an instance is finalized, the finalizers in that instance's inheritance chain are called, in order, from most derived to least derived. A finalizer may be executed on any thread. For further discussion of the rules that govern when and how a finalizer is executed, see §8.9.

[Example: The output of the example

```
using System;
class A
{
   ~A() {
      Console.WriteLine("A's finalizer");
   z
}
class B: A
Ł
   ~B() {
      Console.WriteLine("B's finalizer");
   }
}
class Test
{
   static void Main() {
      B b = new B();
      b = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();
   }
}
B's finalizer
A's finalizer
```

since finalizers in an inheritance chain are called in order, from most derived to least derived. end example]

Finalizers are implemented by overriding the virtual method Finalize on System.Object. C# programs are not permitted to override this method or call it (or overrides of it) directly. [*Example*: For instance, the program

```
class A
{
    override protected void Finalize() {} // error
    public void F() {
        this.Finalize(); // error
    }
}
```

```
contains two errors. end example]
```

The compiler behaves as if this method, and overrides of it, do not exist at all. [*Example*: Thus, this program:

```
class A
{
    void Finalize() {} // permitted
}
```

is valid and the method shown hides System.Object's Finalize method. end example]

For a discussion of the behavior when an exception is thrown from a finalizer, see §21.4.

# 15.14 Iterators

is

# 15.14.1 General

A function member (§12.6) implemented using an iterator block (§13.3) is called an *iterator*.

An iterator block may be used as the body of a function member as long as the return type of the corresponding function member is one of the enumerator interfaces (§15.14.2) or one of the enumerable

interfaces (§15.14.3). It may occur as a *method-body, operator-body* or *accessor-body*, whereas events, instance constructors, static constructors and finalizers may not be implemented as iterators.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any ref or out parameters.

# 15.14.2 Enumerator interfaces

The *enumerator interfaces* are the non-generic interface System.Collections.IEnumerator and all instantiations of the generic interface System.Collections.Generic.IEnumerator<T>. For the sake of brevity, in this subclause and its siblings these interfaces are referenced as IEnumerator and IEnumerator<T>, respectively.

## 15.14.3 Enumerable interfaces

The *enumerable interfaces* are the non-generic interface System.Collections.IEnumerable and all instantiations of the generic interface System.Collections.Generic.IEnumerable<T>. For the sake of brevity, in this subclause and its siblings these interfaces are referenced as IEnumerable and IEnumerable<T>, respectively.

## 15.14.4 Yield type

An iterator produces a sequence of values, all of the same type. This type is called the *yield type* of the iterator.

- The yield type of an iterator that returns IEnumerator or IEnumerable is object.
- The yield type of an iterator that returns IEnumerator<T> or IEnumerable<T> is T.

## 15.14.5 Enumerator objects

#### 15.14.5.1 General

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an *enumerator object* is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's MoveNext method is invoked. An enumerator object has the following characteristics:

- It implements IEnumerator and IEnumerator<T>, where T is the yield type of the iterator.
- It implements System. IDisposable.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
- It has four potential states, *before*, *running*, *suspended*, and *after*, and is initially in the *before* state.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (§7.4.3).

An enumerator object may implement more interfaces than those specified above.

The following subclauses describe the required behavior of the MoveNext, Current, and Dispose members of the IEnumerator and IEnumerator<T> interface implementations provided by an enumerator object.

Enumerator objects do not support the IEnumerator.Reset method. Invoking this method causes a System.NotSupportedException to be thrown.

## 15.14.5.2 The MoveNext method

The MoveNext method of an enumerator object encapsulates the code of an iterator block. Invoking the MoveNext method executes code in the iterator block and sets the Current property of the enumerator object as appropriate. The precise action performed by MoveNext depends on the state of the enumerator object when MoveNext is invoked:

- If the state of the enumerator object is *before*, invoking MoveNext:
  - Changes the state to *running*.
  - Initializes the parameters (including this) of the iterator block to the argument values and instance value saved when the enumerator object was initialized.
  - Executes the iterator block from the beginning until execution is interrupted (as described below).
- If the state of the enumerator object is *running*, the result of invoking MoveNext is unspecified.
- If the state of the enumerator object is *suspended*, invoking MoveNext:
  - Changes the state to *running*.
  - Restores the values of all local variables and parameters (including this) to the values saved when execution of the iterator block was last suspended. [*Note*: The contents of any objects referenced by these variables may have changed since the previous call to MoveNext. *end note*]
  - Resumes execution of the iterator block immediately following the yield return statement that caused the suspension of execution and continues until execution is interrupted (as described below).
- If the state of the enumerator object is *after*, invoking MoveNext returns false.

When MoveNext executes the iterator block, execution can be interrupted in four ways: By a yield return statement, by a yield break statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

- When a yield return statement is encountered (§10.4.4.20):
  - The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the Current property of the enumerator object.
  - Execution of the iterator body is suspended. The values of all local variables and parameters (including this) are saved, as is the location of this yield return statement. If the yield return statement is within one or more try blocks, the associated finally blocks are *not* executed at this time.
  - The state of the enumerator object is changed to *suspended*.
  - The MoveNext method returns true to its caller, indicating that the iteration successfully advanced to the next value.
- When a yield break statement is encountered (§10.4.4.20):
  - If the yield break statement is within one or more try blocks, the associated finally blocks are executed.
  - The state of the enumerator object is changed to *after*.
  - The MoveNext method returns false to its caller, indicating that the iteration is complete.
- When the end of the iterator body is encountered:
  - The state of the enumerator object is changed to *after*.
  - The MoveNext method returns false to its caller, indicating that the iteration is complete.

- When an exception is thrown and propagated out of the iterator block:
  - Appropriate finally blocks in the iterator body will have been executed by the exception propagation.
  - The state of the enumerator object is changed to *after*.
  - The exception propagation continues to the caller of the MoveNext method.

### 15.14.5.3 The Current property

An enumerator object's Current property is affected by yield return statements in the iterator block.

When an enumerator object is in the *suspended* state, the value of Current is the value set by the previous call to MoveNext. When an enumerator object is in the *before*, *running*, or *after* states, the result of accessing Current is unspecified.

For an iterator with a yield type other than object, the result of accessing Current through the enumerator object's IEnumerable implementation corresponds to accessing Current through the enumerator object's IEnumerator<T> implementation and casting the result to object.

#### 15.14.5.4 The Dispose method

The Dispose method is used to clean up the iteration by bringing the enumerator object to the *after* state.

- If the state of the enumerator object is *before*, invoking Dispose changes the state to *after*.
- If the state of the enumerator object is *running*, the result of invoking Dispose is unspecified.
- If the state of the enumerator object is *suspended*, invoking Di spose:
  - Changes the state to *running*.
  - Executes any finally blocks as if the last executed yield return statement were a yield break statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to *after* and the exception is propagated to the caller of the Dispose method.
  - Changes the state to *after*.
- If the state of the enumerator object is *after*, invoking Dispose has no affect.

## 15.14.6 Enumerable objects

#### 15.14.6.1 General

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an *enumerable object* is created and returned. The enumerable object's GetEnumerator method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's MoveNext method is invoked. An enumerable object has the following characteristics:

- It implements IEnumerable and IEnumerable<T>, where T is the yield type of the iterator.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (§7.4.3).

An enumerable object may implement more interfaces than those specified above. [*Note*: For example, an enumerable object may also implement IEnumerator and IEnumerator<br/>and IEnumerator<br/>stor example and an enumerator. Typically, such an implementation would return its own instance (to save allocations) from the first call to GetEnumerator. Subsequent invocations of GetEnumerator, if any, would return a new class instance, typically of the same class, so that calls to different enumerator instances will not affect each other. It cannot return the same instance even if the previous enumerator has already enumerated past the end of the sequence, since all future calls to an exhausted enumerator must throw exceptions. *end note*]

# 15.14.6.2 The GetEnumerator method

An enumerable object provides an implementation of the GetEnumerator methods of the IEnumerable and IEnumerable<T> interfaces. The two GetEnumerator methods share a common implementation that acquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in §15.14.5.

# **15.15 Async Functions**

# 15.15.1 General

A method (§15.6) or anonymous function (§12.16) with the async modifier is called an *async function*. In general, the term *async* is used to describe any kind of function that has the async modifier.

It is a compile-time error for the formal parameter list of an async function to specify any ref or out parameters.

The *return-type* of an async method shall be either void or a *task type*. The task types are System.Threading.Tasks.Task and types constructed from System.Threading.Tasks.Task<T>. For the sake of brevity, in this chapter these types are referenced as Task and Task<T>, respectively. An async method returning a task type is said to be *task-returning*.

The exact definition of the task types is implementation-defined, but from the language's point of view, a task type is in one of the states *incomplete*, *succeeded* or *faulted*. A *faulted* task records a pertinent exception. A *succeeded* Task < T > records a result of type T. Task types are awaitable, and tasks can therefore be the operands of await expressions (§12.8.8).

An async function has the ability to suspend evaluation by means of await expressions (§12.8.8) in its body. Evaluation may later be resumed at the point of the suspending await expression by means of a *resumption delegate*. The resumption delegate is of type System.Action, and when it is invoked, evaluation of the async function invocation will resume from the await expression where it left off. The *current caller* of an async function invocation is the original caller if the function invocation has never been suspended or the most recent caller of the resumption delegate otherwise.

## 15.15.2 Evaluation of a task-returning async function

Invocation of a task-returning async function causes an instance of the returned task type to be generated. This is called the *return task* of the async function. The task is initially in an *incomplete* state.

The async function body is then evaluated until it is either suspended (by reaching an await expression) or terminates, at which point control is returned to the caller, along with the return task.

When the body of the async function terminates, the return task is moved out of the incomplete state:

• If the function body terminates as the result of reaching a return statement or the end of the body, any result value is recorded in the return task, which is put into a *succeeded* state.

• If the function body terminates as the result of an uncaught exception (§13.10.6) the exception is recorded in the return task which is put into a *faulted* state.

# 15.15.3 Evaluation of a void-returning async function

If the return type of the async function is void, evaluation differs from the above in the following way: Because no task is returned, the function instead communicates completion and exceptions to the current thread's **synchronization context**. The exact definition of synchronization context is implementationdependent, but is a representation of "where" the current thread is running. The synchronization context is notified when evaluation of a void-returning async function commences, completes successfully, or causes an uncaught exception to be thrown.

This allows the context to keep track of how many void-returning async functions are running under it, and to decide how to propagate exceptions coming out of them.

# 16. Structs

# 16.1 General

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

[*Note*: Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference. *end note*]

As described in §9.3.5, the simple types provided by C#, such as int, double, and bool, are, in fact, all struct types.

# 16.2 Struct declarations

## 16.2.1 General

A struct-declaration is a type-declaration (§14.7) that declares a new struct:

struct-declaration:

attributes<sub>opt</sub> struct-modifiers<sub>opt</sub> partial<sub>opt</sub> struct identifier type-parameter-list<sub>opt</sub> struct-interfaces<sub>opt</sub> type-parameter-constraints-clauses<sub>opt</sub> struct-body ;<sub>opt</sub>

A struct-declaration consists of an optional set of attributes (§22), followed by an optional set of structmodifiers (§16.2.2), followed by an optional partial modifier (§15.2.7), followed by the keyword struct and an *identifier* that names the struct, followed by an optional *type-parameter-list* specification (§15.2.3), followed by an optional struct-interfaces specification (§16.2.4), followed by an optional *type-parameterconstraints-clauses* specification (§15.2.5), followed by a *struct-body* (§16.2.5), optionally followed by a semicolon.

A struct declaration shall not supply a *type-parameter-constraints-clauses* unless it also supplies a *type-parameter-list*.

A struct declaration that supplies a *type-parameter-list* is a generic struct declaration.

## 16.2.2 Struct modifiers

A struct-declaration may optionally include a sequence of struct modifiers:

struct-modifiers: struct-modifier struct-modifiers struct-modifier struct-modifier:

```
new
public
protected
internal
private
```

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration (§15.2.2).

#### 16.2.3 Partial modifier

The partial modifier indicates that this *struct-declaration* is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in §15.2.7.

## 16.2.4 Struct interfaces

A struct declaration may include a *struct-interfaces* specification, in which case the struct is said to directly implement the given interface types. For a constructed struct type, including a nested type declared within a generic type declaration (§15.3.9.7), each implemented interface type is obtained by substituting, for each *type-parameter* in the given interface, the corresponding *type-argument* of the constructed type.

struct-interfaces:

: interface-type-list

The handling of interfaces on multiple parts of a partial struct declaration (§15.2.7) are discussed further in §15.2.4.3.

Interface implementations are discussed further in §18.6.

## 16.2.5 Struct body

The struct-body of a struct defines the members of the struct.

```
struct-body:
    { struct-member-declarations<sub>opt</sub> }
```

## 16.3 Struct members

The members of a struct consist of the members introduced by its *struct-member-declarations* and the members inherited from the type System.ValueType.

struct-member-declarations: struct-member-declaration struct-member-declarations struct-member-declaration

struct-member-declaration: constant-declaration field-declaration method-declaration property-declaration event-declaration indexer-declaration operator-declaration constructor-declaration static-constructor-declaration type-declaration

[*Note*: All kinds of *class-member-declarations* except *finalizer-declaration* are also *struct-member-declarations*. *end note*] Except for the differences noted in §16.4, the descriptions of class members provided in §15.3 through §15.12 apply to struct members as well.

## 16.4 Class and struct differences

## 16.4.1 General

Structs differ from classes in several important ways:

- Structs are value types (§16.4.2).
- All struct types implicitly inherit from the class System.ValueType (§16.4.3).

- Assignment to a variable of a struct type creates a *copy* of the value being assigned (§16.4.4).
- The default value of a struct is the value produced by setting all fields to their default value (§16.4.5).
- Boxing and unboxing operations are used to convert between a struct type and certain reference types (§16.4.6).
- The meaning of this is different within struct members (§16.4.7).
- Instance field declarations for a struct are not permitted to include variable initializers (§16.4.8).
- A struct is not permitted to declare a parameterless instance constructor (§16.4.9).
- A struct is not permitted to declare a finalizer.

## 16.4.2 Value semantics

Structs are value types (§9.3) and are said to have value semantics. Classes, on the other hand, are reference types (§9.2) and are said to have reference semantics.

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to an object that contains the data. When a struct B contains an instance field of type A and A is a struct type, it is a compile-time error for A to depend on B or a type constructed from B. A struct X *directly depends on* a struct Y if X contains an instance field of type Y. Given this definition, the complete set of structs upon which a struct depends is the transitive closure of the *directly depends on* relationship. [*Example*:

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

is an error because Node contains an instance field of its own type. Another example

```
struct A { B b; }
struct B { C c; }
struct C { A a; }
```

is an error because each of the types A, B, and C depend on each other. end example]

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data (except in the case of ref and out parameter variables), and it is not possible for operations on one to affect the other. Furthermore, except when explicitly nullable (§9.3.11), it is not possible for values of a struct type to be null. [*Note*: If a struct contains a field of reference type then the contents of the object referenced can be altered by other operations. However the value of the field itself, i.e., which object it references, cannot be changed through a mutation of a different struct value. *end note*]

[Example: Given the declaration

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

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

outputs the value 10. The assignment of a to b creates a copy of the value, and b is thus unaffected by the assignment to a.x. Had Point instead been declared as a class, the output would be 100 because a and b would reference the same object. *end example*]

#### 16.4.3 Inheritance

All struct types implicitly inherit from the class System.ValueType, which, in turn, inherits from class object. A struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration to specify a base class.

Struct types are never abstract and are always implicitly sealed. The abstract and sealed modifiers are therefore not permitted in a struct declaration.

Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be protected or protected internal.

Function members in a struct cannot be abstract or virtual, and the override modifier is allowed only to override methods inherited from System.ValueType.

#### 16.4.4 Assignment

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

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a ref or out parameter.

When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access shall be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in §12.18.2.

## 16.4.5 Default values

As described in §10.3, several kinds of variables are automatically initialized to their default value when they are created. For variables of class types and other reference types, this default value is null. However, since structs are value types that cannot be null, the default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to null.

[Example: Referring to the Point struct declared above, the example

Point[] a = new Point[100];

initializes each Point in the array to the value produced by setting the x and y fields to zero. end example]

The default value of a struct corresponds to the value returned by the default constructor of the struct (§9.3.3). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor, which always returns the value that results from setting all fields to their default values.

[Note: Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;
struct KeyValuePair
{
string key;
string value;
```

}

```
public KeyValuePair(string key, string value) {
    if (key == null || value == null) throw new ArgumentException();
    this.key = key;
    this.value = value;
}
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a KeyValuePair variable is subject to default value initialization, the key and value fields will be null, and the struct should be prepared to handle this state. *end note*]

# 16.4.6 Boxing and unboxing

A value of a class type can be converted to type object or to an interface type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a value of type object or a value of an interface type can be converted back to a class type without changing the reference (but, of course, a run-time type check is required in this case).

Since structs are not reference types, these operations are implemented differently for struct types. When a value of a struct type is converted to certain reference types (as defined in §11.2.8), a boxing operation takes place. Likewise, when a value of certain reference types (as defined in §11.3.6) is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing *copies* the struct value either into or out of the boxed instance. [*Note*: Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct. *end note*]

For further details on boxing and unboxing, see §11.2.8 and §11.3.6.

# 16.4.7 Meaning of this

The meaning of this in a struct differs from the meaning of this in a class, as described in §12.7.8. When a struct type overrides a virtual method inherited from System.ValueType (such as Equals, GetHashCode, or ToString), invocation of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. [*Example*:

```
using System;
      struct Counter {
         int value;
         public override string ToString() {
            value++;
            return value.ToString();
         }
      }
      class Program
      ł
         static void Test<T>() where T: new() {
            T x = new T();
            Console.WriteLine(x.ToString());
            Console.WriteLine(x.ToString());
            Console.WriteLine(x.ToString());
         }
         static void Main() {
            Test<Counter>();
         }
      }
The output of the program is:
```

1 2 3 Although it is bad style for ToString to have side effects, the example demonstrates that no boxing occurred for the three invocations of x.ToString(). *end example*]

Similarly, boxing never implicitly occurs when accessing a member on a constrained type parameter when the member is implemented within the value type. For example, suppose an interface ICounter contains a method Increment, which can be used to modify a value. If ICounter is used as a constraint, the implementation of the Increment method is called with a reference to the variable that Increment was called on, never a boxed copy. [*Example*:

```
using System;
interface ICounter
ł
   void Increment();
}
struct Counter: ICounter
{
   int value;
   public override string ToString() {
      return value.ToString();
   }
   void ICounter.Increment() {
      value++;
   }
}
class Program
   static void Test<T>() where T: ICounter, new() {
      T x = new T();
      Console.WriteLine(x);
      x.Increment():
                                     // Modify x
      Console.WriteLine(x);
      ((ICounter)x).Increment();
                                     // Modify boxed copy of x
      Console.WriteLine(x);
   }
   static void Main() {
      Test<Counter>();
   }
}
```

The first call to Increment modifies the value in the variable x. This is not equivalent to the second call to Increment, which modifies the value in a boxed copy of x. Thus, the output of the program is:

0 1 1

end example]

#### 16.4.8 Field initializers

As described in §16.4.5, the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to null. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers. [*Example*: The following

```
struct Point
{
    public int x = 1; // Error, initializer not permitted
    public int y = 1; // Error, initializer not permitted
}
```

is in error because the instance field declarations include variable initializers. end example]

# 16.4.9 Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor, which always returns the value that results from setting all value type fields to their default value and all reference type fields to null (§9.3.3). A struct can declare instance constructors having parameters. [*Example*:

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

Given the above declaration, the statements

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

both create a Point with x and y initialized to zero. end example]

A struct instance constructor is not permitted to include a constructor initializer of the form  $base(argument-list_{opt})$ .

The this parameter of a struct instance constructor corresponds to an out parameter of the struct type. As such, this shall be definitely assigned (§10.4) at every location where the constructor returns. Similarly, it cannot be read (even implicitly) in the constructor body before being definitely assigned.

If the struct instance constructor specifies a constructor initializer, that initializer is considered a definite assignment to this that occurs prior to the body of the constructor. Therefore, the body itself has no initialization requirements. [*Example*: Consider the instance constructor implementation below:

```
struct Point
{
    int x, y;
    public int x {
        set { x = value; }
    }
    public int Y {
        set { y = value; }
    }
    public Point(int x, int y) {
            X = x; // error, this is not yet definitely assigned
            Y = y; // error, this is not yet definitely assigned
    }
}
```

No instance function member (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.

## end example]

## 16.4.10 Static constructors

Static constructors for structs follow most of the same rules as for classes. The execution of a static constructor for a struct type is triggered by the first of the following events to occur within an application domain:

- A static member of the struct type is referenced.
- An explicitly declared constructor of the struct type is called.

[*Note*: The creation of default values (§16.4.5) of struct types does not trigger the static constructor. (An example of this is the initial value of elements in an array.) *end note*]

# 16.4.11 Automatically implemented properties

Automatically implemented properties (§15.7.4) use hidden backing fields, which are only accessible to the property accessors. [*Note*: This access restriction means that constructors in structs containing automatically implemented properties often need an explicit constructor initializer where they would not otherwise need one, to satisfy the requirement of all fields being definitely assigned before any function member is invoked or the constructor returns. *end note*]

# 17. Arrays

# 17.1 General

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

An array has a rank that determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of one is called a *single-dimensional array*. An array with a rank greater than one is called a *multi-dimensional array*. Specific sized multi-dimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so on. Each dimension of an array has an associated length that is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length N, indices can range from 0 to N - 1 inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

Every array type is a reference type (§9.2). The element type of an array can be any type, including value types and array types.

# 17.2 Array types

# 17.2.1 General

The grammar productions for array types are provided in §9.2.1.

An array type is written as a non-array-type followed by one or more rank-specifiers.

A non-array-type is any type that is not itself an array-type.

The rank of an array type is given by the leftmost *rank-specifier* in the *array-type*: A *rank-specifier* indicates that the array is an array with a rank of one plus the number of "," tokens in the *rank-specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank-specifier*:

- An array type of the form T[R] is an array with rank R and a non-array element type T.
- An array type of the form  $T[R][R_1]...[R_N]$  is an array with rank R and an element type  $T[R_1]...[R_N]$ .

In effect, the *rank-specifiers* are read from left to right *before* the final non-array element type. [*Example*: The type int[][,,][,] is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of int. *end example*]

At run-time, a value of an array type can be null or a reference to an instance of that array type. [Note: Following the rules of §17.6, the value may also be a reference to a covariant array type. end note]

# 17.2.2 The System.Array type

The type System.Array is the abstract base type of all array types. An implicit reference conversion (§11.2.7) exists from any array type to System.Array and to any interface type implemented by System.Array. An explicit reference conversion (§11.3.5) exists from System.Array and any interface type implemented by System.Array to any array type. 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.

# 17.2.3 Arrays and the generic collection interfaces

A single-dimensional array T[] implements the interface System.Collections.Generic.IList<T> (IList<T> for short) and its base interfaces. Accordingly, there is an implicit conversion from T[] to IList<T> and its base interfaces. In addition, if there is an implicit reference conversion from S to T then S[] implements IList<T> and there is an implicit reference conversion from S[] to IList<T> and its base interfaces (§11.2.7). If there is an explicit reference conversion from S to T then there is an explicit reference conversion from S[] to IList<T> and its base interfaces (§11.3.5).

Similarly, a single-dimensional array T[] also implements the interface

System.Collections.Generic.IReadOnlyList<T> (IReadOnlyList<T> for short) and its base interfaces. Accordingly, there is an implicit conversion from T[] to IReadOnlyList<T> and its base interfaces. In addition, if there is an implicit reference conversion from S to T then S[] implements IReadOnlyList<T> and there is an implicit reference conversion from S[] to IReadOnlyList<T> and its base interfaces (§11.2.7). If there is an explicit reference conversion from S to T then there is an explicit reference conversion from S to T then there is an explicit reference conversion from S to T then there is an explicit reference conversion from S[] to IReadOnlyList<T> and its base interfaces (§11.3.5).

[Example: For example:

```
using System.Collections.Generic;
class Test
{
    static void Main()
    {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;
                                                   // Ok
        IList<string> lst1 = sa;
        IList<string> lst2 = oa1;
                                                   // Error, cast needed
        IList<object> lst3 = sa;
                                                    // Ok
        IList<object> lst4 = oa1;
                                                    // Ok
        IList<string> lst5 = (IList<string>)oal;
                                                    // Exception
        IList<string> lst6 = (IList<string>)oa2;
                                                    // Ok
        IReadOnlyList<string> lst7 = sa;
                                                    // Ok
        IReadOnlyList<string> lst8 = oa1;
                                                    // Error, cast needed
        IReadOnlyList<object> lst9 = sa;
                                                    // Ok
        IReadOnlyList<object> lst10 = oa1;
                                                    // Ok
        IReadOnlyList<string> lst11 = (IReadOnlyList<string>)oa1;
                                                     // Exception
        IReadOnlyList<string> lst12 = (IReadOnlyList<string>)oa2; // Ok
    }
}
```

The assignment lst2 = oal generates a compile-time error since the conversion from object[] to IList<string> is an explicit conversion, not implicit. The cast (IList<string>)oal will cause an exception to be thrown at run-time since oal references an object[] and not a string[]. However the cast (IList<string>)oa2 will not cause an exception to be thrown since oa2 references a string[]. end example]

Whenever there is an implicit or explicit reference conversion from S[] to IList<T>, there is also an explicit reference conversion from IList<T> and its base interfaces to S[] (§11.3.5).

When an array type S[] implements IList<T>, some of the members of the implemented interface may throw exceptions. The precise behavior of the implementation of the interface is beyond the scope of this specification.

# 17.3 Array creation

Array instances are created by *array-creation-expressions* (§12.7.11.5) or by field or local variable declarations that include an *array-initializer* (§17.7). Array instances can also be created implicitly as part of evaluating an argument list involving a parameter array (§15.6.2.5).

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 (§10.3).

# 17.4 Array element access

Array elements are accessed using *element-access* expressions (\$12.7.7.2) of the form A[I<sub>1</sub>, I<sub>2</sub>, ..., I<sub>N</sub>], where A is an expression of an array type and each I<sub>x</sub> is an expression of type int, uint, long, ulong, or can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a foreach statement (§13.9.5).

# 17.5 Array members

Every array type inherits the members declared by the System.Array type.

# 17.6 Array covariance

For any two *reference-types* A and B, if an implicit reference conversion (§11.2.7) or explicit reference conversion (§11.3.4) exists from A to B, then the same reference conversion also exists from the array type A[R] to the array type B[R], where R is any given *rank-specifier* (but the same for both array types). This relationship is known as *array covariance*. Array covariance, in particular, means that a value of an array type A[R] might 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 (§12.18.2). [*Example*:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }
    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}</pre>
```

The assignment to array[i] in the Fill method implicitly includes a run-time check, which ensures that value is either a null reference or a reference to an object of a type that is compatible with the actual element type of array. In Main, the first two invocations of Fill succeed, but the third invocation causes a System.ArrayTypeMismatchException to be thrown upon executing the first assignment to array[i]. The exception occurs because a boxed int cannot be stored in a string array. *end example*]

Array covariance specifically does not extend to arrays of *value-types*. For example, no conversion exists that permits an int[] to be treated as an object[].

# 17.7 Array initializers

Array initializers may be specified in field declarations (§15.5), local variable declarations (§13.6.2), and array creation expressions (§12.7.11.5):

```
array-initializer:

{ variable-initializer-list<sub>opt</sub> }

{ variable-initializer-list , }

variable-initializer-list:

variable-initializer

variable-initializer

variable-initializer:
```

expression array-initializer

An array initializer consists of a sequence of variable initializers, enclosed by "{"and "}" tokens and separated by "," tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer, or is inferred from the expressions in the array initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, [*Example*:

int[]  $a = \{0, 2, 4, 6, 8\};$ 

end example] it is simply shorthand for an equivalent array creation expression: [Example:

int[] a = new int[] {0, 2, 4, 6, 8};

end example]

For a single-dimensional array, the array initializer shall consist of a sequence of expressions, each having an implicit conversion to the element type of the array (§11.2). The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. [*Example*: The array initializer above creates an int[] instance of length 5 and then initializes the instance with the following values:

a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;

#### end example]

For a multi-dimensional array, the array initializer shall have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements shall be the same as the other array initializers at the same level. [*Example*: The example:

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:

int[,] b = new int[5, 2];

and then initializes the array instance with the following values:

b[0, 0] = 0; b[0, 1] = 1; b[1, 0] = 2; b[1, 1] = 3; b[2, 0] = 4; b[2, 1] = 5; b[3, 0] = 6; b[3, 1] = 7; b[4, 0] = 8; b[4, 1] = 9;

#### end example]

If a dimension other than the rightmost is given with length zero, the subsequent dimensions are assumed to also have length zero. [*Example*:

int[,] c = {};

creates a two-dimensional array with a length of zero for both the leftmost and the rightmost dimension:

int[,] c = new int[0, 0];

end example]

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths shall be constant expressions and the number of elements at each nesting level shall match the corresponding dimension length. [*Example*: Here are some examples:

int i = 3; int[] x = new int[3] {0, 1, 2}; // OK int[] y = new int[i] {0, 1, 2}; // Error, i not a constant int[] z = new int[3] {0, 1, 2, 3}; // Error, length/initializer mismatch

Here, the initializer for y results in a compile-time error because the dimension length expression is not a constant, and the initializer for z results in a compile-time error because the length and the number of elements in the initializer do not agree. *end example*]

[*Note*: C# allows a trailing comma at the end of an *array-initializer*. This syntax provides flexibility in adding or deleting members from such a list, and simplifies machine generation of such lists. *end note*]

# 18. Interfaces

# 18.1 General

An interface defines a contract. A class or struct that implements an interface shall adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it declares. The interface merely specifies the members that shall be supplied by classes or structs that implement the interface.

# **18.2 Interface declarations**

# 18.2.1 General

An interface-declaration is a type-declaration (§14.7) that declares a new interface type.

interface-declaration: attributes<sub>opt</sub> interface-modifiers<sub>opt</sub> partial<sub>opt</sub> interface identifier variant-type-parameter-list<sub>opt</sub> interface-base<sub>opt</sub> type-parameter-constraints-clauses<sub>opt</sub> interface-body ;<sub>opt</sub>

An *interface-declaration* consists of an optional set of *attributes* (§22), followed by an optional set of *interface-modifiers* (§18.2.2), followed by an optional partial modifier (§15.2.7), followed by the keyword interface and an *identifier* that names the interface, followed by an optional *variant-type-parameter-list* specification (§18.2.3), followed by an optional *interface-base* specification (§18.2.4), followed by an optional *type-parameter-constraints-clauses* specification (§15.2.5), followed by an *interface-body* (§18.3), optionally followed by a semicolon.

An interface declaration shall not supply a *type-parameter-constraints-clauses* unless it also supplies a *type-parameter-list*.

An interface declaration that supplies a *type-parameter-list* is a generic interface declaration.

## 18.2.2 Interface modifiers

An interface-declaration may optionally include a sequence of interface modifiers:

interface-modifiers: interface-modifier interface-modifiers interface-modifier interface-modifier: new public protected internal private

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The new modifier is only permitted on interfaces defined within a class. It specifies that the interface hides an inherited member by the same name, as described in §15.3.5.

The public, protected, internal, and private modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers might be

permitted (§8.5.2). When a partial type declaration (§15.2.7) includes an accessibility specification (via the public, protected, internal, and private modifiers), the rules in §15.2.2 apply.

# 18.2.3 Variant type parameter lists

### 18.2.3.1 General

Variant type parameter lists can only occur on interface and delegate types. The difference from ordinary *type-parameter-lists* is the optional *variance-annotation* on each type parameter.

If the variance annotation is out, the type parameter is said to be *covariant*. If the variance annotation is in, the type parameter is said to be *contravariant*. If there is no variance annotation, the type parameter is said to be *invariant*.

#### [Example: In the following:

```
interface C<out X, in Y, Z>
{
    X M(Y y);
    Z P { get; set; }
}
```

X is covariant, Y is contravariant and Z is invariant. end example]

If a generic interface is declared in multiple parts (§15.2.3), each partial declaration shall specify the same variance for each type parameter.

## 18.2.3.2 Variance safety

The occurrence of variance annotations in the type parameter list of a type restricts the places where types can occur within the type declaration.

A type T is *output-unsafe* if one of the following holds:

- T is a contravariant type parameter
- T is an array type with an output-unsafe element type
- T is an interface or delegate type S<A<sub>1</sub>,... A<sub>K</sub>> constructed from a generic type S<X<sub>1</sub>, ... X<sub>K</sub>> where for at least one A<sub>1</sub> one of the following holds:
  - $\circ$  X<sub>i</sub> is covariant or invariant and A<sub>i</sub> is output-unsafe.
  - $\circ$  X<sub>i</sub> is contravariant or invariant and A<sub>i</sub> is input-unsafe.

#### A type T is *input-unsafe* if one of the following holds:

- T is a covariant type parameter
- T is an array type with an input-unsafe element type
- T is an interface or delegate type S<A<sub>1</sub>,... A<sub>K</sub>> constructed from a generic type S<X<sub>1</sub>, ... X<sub>K</sub>> where for at least one A<sub>1</sub> one of the following holds:
  - $\circ$  X<sub>i</sub> is covariant or invariant and A<sub>i</sub> is input-unsafe.

 $\circ$  X<sub>i</sub> is contravariant or invariant and A<sub>i</sub> is output-unsafe.

Intuitively, an output-unsafe type is prohibited in an output position, and an input-unsafe type is prohibited in an input position.

A type is *output-safe* if it is not output-unsafe, and *input-safe* if it is not input-unsafe.

## 18.2.3.3 Variance conversion

The purpose of variance annotations is to provide for more lenient (but still type safe) conversions to interface and delegate types. To this end the definitions of implicit (§11.2) and explicit conversions (§11.3) make use of the notion of variance-convertibility, which is defined as follows:

A type T<A<sub>1</sub>, ..., A<sub>n</sub>> is variance-convertible to a type T<B<sub>1</sub>, ..., B<sub>n</sub>> if T is either an interface or a delegate type declared with the variant type parameters T<X<sub>1</sub>, ..., X<sub>n</sub>>, and for each variant type parameter X<sub>1</sub> one of the following holds:

- X<sub>i</sub> is covariant and an implicit reference or identity conversion exists from A<sub>i</sub> to B<sub>i</sub>
- X<sub>i</sub> is contravariant and an implicit reference or identity conversion exists from B<sub>i</sub> to A<sub>i</sub>
- $X_i$  is invariant and an identity conversion exists from  $A_i$  to  $B_i$

## 18.2.4 Base interfaces

An interface can inherit from zero or more interface types, which are called the *explicit base interfaces* of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma-separated list of base interface types.

interface-base:

: interface-type-list

The explicit base interfaces can be constructed interface types (§9.4, §18.2). A base interface cannot be a type parameter on its own, though it can involve the type parameters that are in scope.

For a constructed interface type, the explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each *type-parameter* in the base interface declaration, the corresponding *type-argument* of the constructed type.

The explicit base interfaces of an interface shall be at least as accessible as the interface itself (§8.5.5). [*Note*: For example, it is a compile-time error to specify a private or internal interface in the *interface-base* of a public interface. *end note*]

It is a compile-time error for an interface to directly or indirectly inherit from itself.

The **base interfaces** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. [*Example*: In the following code

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

the base interfaces of IComboBox are IControl, ITextBox, and IListBox. In other words, the IComboBox interface above inherits members SetText and SetItems as well as Paint. *end example*]

Members inherited from a constructed generic type are inherited after type substitution. That is, any constituent types in the member have the base class declaration's type parameters replaced with the corresponding type arguments used in the *class-base* specification. [*Example*: In the following code

```
interface IBase<T>
{
    T[] Combine(T a, T b);
}
interface IDerived : IBase<string[,]>
{
    // Inherited: string[][,] Combine(string[,] a, string[,] b);
}
```

the interface IDerived inherits the Combine method after the type parameter T is replaced with string[,]. *end example*]

A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

The handling of interfaces on multiple parts of a partial interface declaration (§15.2.7) are discussed further in §15.2.4.3.

Every base interface of an interface shall be output-safe (§18.2.3.2).

# 18.3 Interface body

The *interface-body* of an interface defines the members of the interface.

```
interface-body:
    { interface-member-declarations<sub>opt</sub> }
```

## **18.4 Interface members**

## 18.4.1 General

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

interface-member-declarations: interface-member-declaration interface-member-declarations interface-member-declaration

interface-member-declaration: interface-method-declaration interface-property-declaration interface-event-declaration interface-indexer-declaration

An interface declaration declares zero or more members. The members of an interface shall be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, finalizers, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers.

An *interface-declaration* creates a new declaration space (§8.3), and the type parameters and *interface-member-declarations* immediately contained by the *interface-declaration* introduce new members into this declaration space. The following rules apply to *interface-member-declarations*:

- The name of a type parameter in the *type-parameter-list* of an interface declaration shall differ from the names of all other type parameters in the same *type-parameter-list* and shall differ from the names of all members of the interface.
- The name of a method shall differ from the names of all properties and events declared in the same interface. In addition, the signature (§8.6) of a method shall differ from the signatures of all other methods declared in the same interface, and two methods declared in the same interface may not have signatures that differ solely by ref and out.
- The name of a property or event shall differ from the names of all other members declared in the same interface.
- The signature of an indexer shall differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to *hide* the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member shall include a new modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in §8.7.2.3.

If a new modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

[*Note*: The members in class object are not, strictly speaking, members of any interface (§18.4). However, the members in class object are available via member lookup in any interface type (§12.5). *end note*]

The set of members of an interface declared in multiple parts (§15.2.7) is the union of the members declared in each part. The bodies of all parts of the interface declaration share the same declaration space (§8.3), and the scope of each member (§8.7) extends to the bodies of all the parts.

# 18.4.2 Interface methods

Interface methods are declared using *interface-method-declarations*:

```
interface-method-declaration:
    attributes<sub>opt</sub> new<sub>opt</sub> return-type identifier type-parameter-list<sub>opt</sub>
    ( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub> ;
```

The *attributes*, *return-type*, *identifier*, and *formal-parameter-list* of an interface method declaration have the same meaning as those of a method declaration in a class (§15.6). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon. An *interface-method-declaration* shall not have *type-parameter-constraints-clauses* unless it also has a *type-parameter-list*.

Each formal parameter type of an interface method shall be input-safe (§18.2.3.2), and the return type shall be either void or output-safe. In addition, any out or ref formal parameter types shall also be output-safe. [*Note*: Even out parameters are required to be input-safe, due to common implementation restrictions. *end note*] Furthermore, each class type constraint, interface type constraint and type parameter constraint on any type parameter of the method shall be input-safe.

These rules ensure that any covariant or contravariant usage of the interface remains typesafe. [Example:

interface I<out T> { void M<U>() where U : T; }

is ill-formed because the usage of T as a type parameter constraint on U is not input-safe.

Were this restriction not in place it would be possible to violate type safety in the following manner:

```
class B {}
class D : B {}
class D : B {}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

This is actually a call to C.M<E>. But that call requires that E derive from D, so type safety would be violated here. *end example*]

#### 18.4.3 Interface properties

Interface properties are declared using interface-property-declarations:

```
interface-property-declaration:
    attributes<sub>opt</sub> new<sub>opt</sub> type identifier { interface-accessors }
interface-accessors:
    attributes<sub>opt</sub> get ;
    attributes<sub>opt</sub> get ;
    attributes<sub>opt</sub> get ; attributes<sub>opt</sub> set ;
    attributes<sub>opt</sub> set ; attributes<sub>opt</sub> get ;
```

The *attributes, type,* and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class (§15.7).

The accessors of an interface property declaration correspond to the accessors of a class property declaration (§15.7.3), except that the accessor body shall always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

The type of an interface property shall be output-safe if there is a get accessor, and shall be input-safe if there is a set accessor.

#### 18.4.4 Interface events

Interface events are declared using interface-event-declarations:

The *attributes, type,* and *identifier* of an interface event declaration have the same meaning as those of an event declaration in a class (§15.8).

The type of an interface event shall be input-safe.

#### 18.4.5 Interface indexers

Interface indexers are declared using interface-indexer-declarations:

```
interface-indexer-declaration:
    attributes<sub>opt</sub> new<sub>opt</sub> type this [ formal-parameter-list ] { interface-accessors }
```

The *attributes*, *type*, and *formal-parameter-list* of an interface indexer declaration have the same meaning as those of an indexer declaration in a class (§15.9).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (§15.9), except that the accessor body shall always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

All the formal parameter types of an interface indexer shall be input-safe. In addition, any out or ref formal parameter types shall also be output-safe. [*Note*: Even out parameters are required to be input-safe, due to common implementation restrictions. *end note*]

The type of an interface indexer shall be output-safe if there is a get accessor, and shall be input-safe if there is a set accessor.

# 18.4.6 Interface member access

Interface members are accessed through member access (§12.7.5) and indexer access (§12.7.7.3) expressions of the form I.M and I[A], where I is an interface type, M is a method, property, or event of that interface type, and A is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup (§12.5), method invocation (§12.7.6.2), and indexer access (§12.7.7.3) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This subclause shows several examples, some of which lead to ambiguities and others which don't. In all cases, explicit casts can be used to resolve the ambiguities.

[Example: In the following code

```
interface IList
Ł
   int Count { get; set; }
}
interface ICounter
Ł
   void Count(int i);
}
interface IListCounter: IList, ICounter {}
class C
   void Test(IListCounter x) {
      x.Count(1);
x.Count = 1;
                                      Error
                                      Error
                                    // Ok, invokes IList.Count.set
      ((IList)x).Count = 1;
      ((ICounter)x).Count(1);
                                    // Ok, invokes ICounter.Count
   }
}
```

the first two statements cause compile-time errors because the member lookup (§12.5) of Count in IListCounter is ambiguous. As illustrated by the example, the ambiguity is resolved by casting x to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time. *end example*]

[Example: In the following code

```
interface IInteger
{
    void Add(int i);
}
interface IDouble
{
    void Add(double d);
}
interface INumber: IInteger, IDouble {}
```

the invocation n.Add(1) selects IInteger.Add by applying overload resolution rules of §12.6.4. Similarly, the invocation n.Add(1.0) selects IDouble.Add. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity. *end example*]

[Example: In the following code

```
interface IBase
{
   void F(int i);
}
interface ILeft: IBase
{
   new void F(int i);
}
interface IRight: IBase
{
   void G();
}
interface IDerived: ILeft, IRight {}
class A
   void Test(IDerived d) {
                                 Invokes ILeft.F
      d.F(1);
       ((IBase)d) F(1);
                                 Invokes IBase.F
Invokes ILeft.F
       ((ILeft)d).F(1);
       ((IRight)d).F(1);
                              // Invokes IBase.F
   }
}
```

the IBase.F member is hidden by the ILeft.F member. The invocation d.F(1) thus selects ILeft.F, even though IBase.F appears to not be hidden in the access path that leads through IRight.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from IDerived to ILeft to IBase hides IBase. F, the member is also hidden in the access path from IDerived to IRight to IBase. end example]

# 18.5 Qualified interface member names

An interface member is sometimes referred to by its **qualified interface member name**. The qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The qualified name of a member references the interface in which the member is declared. [*Example*: Given the declarations

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

the qualified name of Paint is IControl.Paint and the qualified name of SetText is ITextBox.SetText. In the example above, it is not possible to refer to Paint as ITextBox.Paint.end example]

When an interface is part of a namespace, a qualified interface member name can include the namespace name. [*Example*:

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

Within the System namespace, both ICloneable.Clone and System.ICloneable.Clone are qualified interface member names for the Clone method. *end example*]

# **18.6 Interface implementations**

# 18.6.1 General

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface is included in the base class list of the class or struct. [*Example*:

```
interface ICloneable
{
    object Clone();
}
interface IComparable
{
    int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

## end example]

A class or struct that directly implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. [*Example*:

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Here, class TextBox implements both IControl and ITextBox. end example]

When a class C directly implements an interface, all classes derived from C also implement the interface implicitly.

The base interfaces specified in a class declaration can be constructed interface types (§9.4, §18.2). [*Example*: The following code illustrates how a class can implement constructed interface types:

```
class C<U,V> {}
interface I1<V> {}
class D: C<string,int>, I1<string> {}
class E<T>: C<int,T>, I1<T> {}
```

end example]

The base interfaces of a generic class declaration shall satisfy the uniqueness rule described in §18.6.3.

# 18.6.2 Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare *explicit interface member implementations*. An explicit interface member implementation is a method, property, event, or indexer declaration that references a qualified interface member name. [*Example*:

```
interface IList<T>
{
    T[] GetElements();
}
interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}
class List<T>: IList<T>, IDictionary<int,T>
{
    public T[] GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

Here IDictionary<int,T>.this and IDictionary<int,T>.Add are explicit interface member implementations. *end example*]

[*Example*: In some cases, the name of an interface member might not be appropriate for the implementing class, in which case, the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a Close member function that has the effect of releasing the file resource, and implement the Dispose method of the IDisposable interface using explicit interface member implementation:

```
interface IDisposable
{
    void Dispose();
}
class MyFile: IDisposable
{
    void IDisposable.Dispose()
    {
        Close();
    }
    public void Close()
    {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

#### end example]

It is not possible to access an explicit interface member implementation through its qualified interface member name in a method invocation, property access, event access, or indexer access. An explicit interface member implementation can only be accessed through an interface instance, and is in that case referenced simply by its member name.

It is a compile-time error for an explicit interface member implementation to include any modifiers (§15.6) other than extern or async.

It is a compile-time error for an explicit interface method implementation to include *type-parameter-constraints-clauses*. The constraints for a generic explicit interface method implementation are inherited from the interface method.[*Note*: Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through a qualified interface member name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through the interface, they are in a sense also as public as the interface in which they are declared.

Explicit interface member implementations serve two primary purposes:

- Because explicit interface member implementations are not accessible through class or struct instances, they allow interface implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of that class or struct.
- Explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

#### end note]

For an explicit interface member implementation to be valid, the class or struct shall name an interface in its base class list that contains a member whose qualified interface member name, type, number of type parameters, and parameter types exactly match those of the explicit interface member implementation. If an interface function member has a parameter array, the corresponding parameter of an associated explicit interface member implementation is allowed, but not required, to have the params modifier. If the interface function member does not have a parameter array then an associated explicit interface member implementation shall not have a parameter array. [*Example*: Thus, in the following class

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...} // invalid
}
```

the declaration of IComparable.CompareTo results in a compile-time error because IComparable is not listed in the base class list of Shape and is not a base interface of ICloneable. Likewise, in the declarations

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}
class Ellipse: Shape
{
    object ICloneable.Clone() {...} // invalid
}
```

the declaration of ICloneable.Clone in Ellipse results in a compile-time error because ICloneable is not explicitly listed in the base class list of Ellipse. *end example*]

The qualified interface member name of an explicit interface member implementation shall reference the interface in which the member was declared. [*Example*: Thus, in the declarations

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

the explicit interface member implementation of Paint must be written as IControl.Paint, not ITextBox.Paint.end example]

## 18.6.3 Uniqueness of implemented interfaces

The interfaces implemented by a generic type declaration shall remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. [*Example*: Suppose a generic class declaration were permitted to be written as follows:

```
interface I<T>
{
    void F();
}
class X<U,V>: I<U>, I<V> // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Were this permitted, it would be impossible to determine which code to execute in the following case:

```
I<int> x = new X<int,int>();
x.F();
```

end example]

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

- Let L be the list of interfaces directly specified in a generic class, struct, or interface declaration C.
- Add to L any base interfaces of the interfaces already in L.
- Remove any duplicates from L.
- If any possible constructed type created from C would, after type arguments are substituted into L, cause two interfaces in L to be identical, then the declaration of C is invalid. Constraint declarations are not considered when determining all possible constructed types.

[*Note*: In the class declaration X above, the interface list L consists of I<U> and I<V>. The declaration is invalid because any constructed type with U and V being the same type would cause these two interfaces to be identical types. *end note*]

It is possible for interfaces specified at different inheritance levels to unify:

```
interface I<T>
{
    void F();
}
class Base<U>: I<U>
{
    void I<U>.F() {...}
}
class Derived<U,V>: Base<U>, I<V> // Ok
{
    void I<V>.F() {...}
}
```

This code is valid even though Derived<U, V> implements both I<U> and I<V>. The code

```
I<int> x = new Derived<int,int>();
x.F();
```

invokes the method in Derived, since Derived<int, int> effectively re-implements I<int> (§18.6.7).

## 18.6.4 Implementation of generic methods

When a generic method implicitly implements an interface method, the constraints given for each method type parameter shall be equivalent in both declarations (after any interface type parameters are replaced with the appropriate type arguments), where method type parameters are identified by ordinal positions, left to right. [*Example*: In the following code:

```
interface I<X, Y, Z>
{
    void F<T>(T t) where T: X;
    void G<T>(T t) where T: Y;
    void H<T>(T t) where T: Z;
}
class C: I<object,C,string>
{
    public void F<T>(T t) {...} // Ok
    public void G<T>(T t) where T: C {...} // Ok
    public void H<T>(T t) where T: string {...} // Error
}
```

the method C.F<T> implicitly implements I<object,C,string>.F<T>. In this case, C.F<T> is not required (nor permitted) to specify the constraint T: object since object is an implicit constraint on all type parameters. The method C.G<T> implicitly implements I<object,C,string>.G<T> because the constraints match those in the interface, after the interface type parameters are replaced with the corresponding type arguments. The constraint for method C.H<T> is an error because sealed types (string in this case) cannot be used as constraints. Omitting the constraint would also be an error since constraints of implicit interface method implementations are required to match. Thus, it is impossible to implicitly implement I<object,C,string>.H<T>. This interface method can only be implemented using an explicit interface member implementation:

```
class C: I<object,C,string>
{
    ...
    public void H<U>(U u) where U: class {...}
    void I<object,C,string>.H<T>(T t) {
        string s = t; // Ok
        H<T>(t);
    }
}
```

In this case, the explicit interface member implementation invokes a public method having strictly weaker constraints. The assignment from t to s is valid since T inherits a constraint of T: string, even though this constraint is not expressible in source code. *end example*]

[*Note*: When a generic method explicitly implements an interface method no constraints are allowed on the implementing method (§15.7.1, §18.6.2) *end note*].

# 18.6.5 Interface mapping

A class or struct shall provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as **interface mapping**.

Interface mapping for a class or struct C locates an implementation for each member of each interface specified in the base class list of C. The implementation of a particular interface member I.M, where I is the interface in which the member M is declared, is determined by examining each class or struct S, starting with C and repeating for each successive base class of C, until a match is located:

- If S contains a declaration of an explicit interface member implementation that matches I and M, then this member is the implementation of I.M.
- Otherwise, if S contains a declaration of a non-static public member that matches M, then this member is the implementation of I.M. If more than one member matches, it is unspecified which member is the implementation of I.M. This situation can only occur if S is a constructed type where the two members as declared in the generic type have different signatures, but the type arguments make their signatures identical.

A compile-time error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of C. The members of an interface include those members that are inherited from base interfaces.

Members of a constructed interface type are considered to have any type parameters replaced with the corresponding type arguments as specified in §15.3.3. [*Example*: For example, given the generic interface declaration:

```
interface I<T>
{
    T F(int x, T[,] y);
    T this[int y] { get; }
}
```

the constructed interface I<string[]> has the members:

```
string[] F(int x, string[,][] y);
string[] this[int y] { get; }
```

## end example]

For purposes of interface mapping, a class or struct member A matches an interface member B when:

- A and B are methods, and the name, type, and formal parameter lists of A and B are identical.
- A and B are properties, the name and type of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).
- A and B are events, and the name and type of A and B are identical.
- A and B are indexers, the type and formal parameter lists of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface-mapping algorithm are:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Neither non-public nor static members participate in interface mapping.

[*Example*: In the following code

```
interface ICloneable
{
    object Clone();
}
class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

the **ICloneable.Clone** member of C becomes the implementation of **Clone** in **ICloneable** because explicit interface member implementations take precedence over other members. *end example*]

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. [*Example*:

```
interface IControl
{
    void Paint();
}
interface IForm
{
    void Paint();
}
class Page: IControl, IForm
{
    public void Paint() {...}
}
```

Here, the Paint methods of both IControl and IForm are mapped onto the Paint method in Page. It is of course also possible to have separate explicit interface member implementations for the two methods. *end example*]

If a class or struct implements an interface that contains hidden members, then some members may need to be implemented through explicit interface member implementations. [*Example*:

```
interface IBase
{
    int P { get; }
}
interface IDerived: IBase
{
    new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}
class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}
class C: IDerived
{
    int IBase.P { get {...} }
}
```

public int P() {...}

#### end example]

}

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. [*Example*: In the following code

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

it is not possible to have separate implementations for the IControl named in the base class list, the IControl inherited by ITextBox, and the IControl inherited by IListBox. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of ITextBox and IListBox share the same implementation of IControl, and ComboBox is simply considered to implement three interfaces, IControl, ITextBox, and IListBox. end example]

The members of a base class participate in interface mapping. [Example: In the following code

```
interface Interface1
{
    void F();
}
class Class1
{
    public void F() {}
    public void G() {}
}
class Class2: Class1, Interface1
{
    new public void G() {}
}
```

the method F in Class1 is used in Class2's implementation of Interface1. end example]

## 18.6.6 Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly *re-implementing* an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. [*Example*: In the declarations

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

```
class Control: IControl
{
    public void Paint() {...}
}
class TextBox: Control
{
    new public void Paint() {...}
}
```

the Paint method in TextBox hides the Paint method in Control, but it does not alter the mapping of Control.Paint onto IControl.Paint, and calls to Paint through class instances and interface instances will have the following effects

#### end example]

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. [*Example*: Rewriting the declarations above to

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    public virtual void Paint() {...}
}
class TextBox: Control
{
    public override void Paint() {...}
}
```

the following effects will now be observed

end example]

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. [*Example*:

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
```

```
protected virtual void PaintControl() {...}
}
class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Here, classes derived from Control can specialize the implementation of IControl.Paint by overriding the PaintControl method. end example]

## 18.6.7 Interface re-implementation

A class that inherits an interface implementation is permitted to *re-implement* the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. [*Example*: In the declarations

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    void IControl.Paint() {...}
}
class MyControl: Control, IControl
{
    public void Paint() {}
}
```

the fact that Control maps IControl.Paint onto Control.IControl.Paint doesn't affect the reimplementation in MyControl, which maps IControl.Paint onto MyControl.Paint. end example]

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. [*Example*:

```
interface IMethods
ł
   void F():
   void G();
   void H();
   void I();
}
class Base: IMethods
   void IMethods.F()
   void IMethods.G()
public void H() {}
                        { }
   public void I() {}
}
class Derived: Base, IMethods
   public void F() {}
   void IMethods.H() {}
}
```

Here, the implementation of IMethods in Derived maps the interface methods onto Derived.F, Base.IMethods.G, Derived.IMethods.H, and Base.I. end example]

When a class implements an interface, it implicitly also implements all that interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. [*Example*:

```
interface IBase
{
   void F();
}
interface IDerived: IBase
{
   void G();
}
class C: IDerived
   void IBase.F() {...}
   void IDerived.G() {...}
}
class D: C, IDerived
   public void F() {...}
   public void G() {...}
}
```

Here, the re-implementation of IDerived also re-implements IBase, mapping IBase. F onto D.F. end example]

### 18.6.8 Abstract classes and interfaces

Like a non-abstract class, an abstract class shall provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. [*Example*:

```
interface IMethods
{
    void F();
    void G();
}
abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Here, the implementation of IMethods maps F and G onto abstract methods, which shall be overridden in non-abstract classes that derive from C. *end example*]

Explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. [*Example*:

```
interface IMethods
{
    void F();
    void G();
}
abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

Here, non-abstract classes that derive from C would be required to override FF and GG, thus providing the actual implementation of IMethods. *end example*]

# 19. Enums

# 19.1 General

An enum type is a distinct value type (§9.2) that declares a set of named constants. [Example: The example

```
enum Color
{
Red,
Green,
Blue
}
```

declares an enum type named Color with members Red, Green, and Blue. end example]

### 19.2 Enum declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword enum, and defines the name, accessibility, underlying type, and members of the enum.

```
enum-declaration:
attributes<sub>opt</sub> enum-modifiers<sub>opt</sub> enum identifier enum-base<sub>opt</sub> enum-body ;<sub>opt</sub>
enum-base:
```

: integral-type

enum-body:

{ enum-member-declarations<sub>opt</sub> }

{ enum-member-declarations , }

Each enum type has a corresponding integral type called the *underlying type* of the enum type. This underlying type shall be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of byte, sbyte, short, ushort, int, uint, long or ulong. [*Note*: char cannot be used as an underlying type. *end note*] An enum declaration that does not explicitly declare an underlying type has an underlying type of int.

[Example: The example

```
enum Color: long
{
Red,
Green,
Blue
}
```

declares an enum with an underlying type of long. *end example*] [*Note*: A developer might choose to use an underlying type of long, as in the example, to enable the use of values that are in the range of long but not in the range of int, or to preserve this option for the future. *end note*]

[Note: C# allows a trailing comma in an *enum-body*, just like it allows one in an *array-initializer* (§17.7). *end* note]

## 19.3 Enum modifiers

An enum-declaration may optionally include a sequence of enum modifiers:

enum-modifiers: enum-modifier enum-modifiers enum-modifier

```
enum-modifier:
new
public
protected
internal
private
```

It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

The modifiers of an enum declaration have the same meaning as those of a class declaration (§15.2.2). However, the abstract, and sealed, and static modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

## 19.4 Enum members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

```
enum-member-declarations:
enum-member-declaration
enum-member-declarations , enum-member-declaration
enum-member-declaration:
attributes<sub>opt</sub> identifier
attributes<sub>opt</sub> identifier = constant-expression
```

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member shall be in the range of the underlying type for the enum. [*Example*: The example

```
enum Color: uint
{
Red = -1,
Green = -2,
Blue = -3
}
```

results in a compile-time error because the constant values -1, -2, and -3 are not in the range of the underlying integral type uint. *end example*]

Multiple enum members may share the same associated value. [Example: The example

```
enum Color
{
Red,
Green,
Blue,
Max = Blue
}
```

shows an enum in which two enum members—Blue and Max—have the same associated value. *end example*]

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant-expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is zero.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by one. This increased value shall be within the range of values that can be represented by the underlying type, otherwise a compile-time error occurs.

```
[Example: The example
```

```
using System;
enum Color
{
   Red.
   Green = 10,
   Blue
}
class Test
   static void Main() {
       Console.WriteLine(StringFromColor(Color.Red));
Console.WriteLine(StringFromColor(Color.Green));
Console.WriteLine(StringFromColor(Color.Blue));
   }
   static string StringFromColor(Color c) {
       switch (c)
           case Color.Red:
               return String.Format("Red = {0}", (int) c);
           case Color.Green:
               return String.Format("Green = {0}", (int) c);
           case Color.Blue:
               return String.Format("Blue = {0}", (int) c);
           default:
               return "Invalid color";
       }
   }
}
```

prints out the enum member names and their associated values. The output is:

Red = 0 Green = 10 Blue = 11

for the following reasons:

- the enum member Red is automatically assigned the value zero (since it has no initializer and is the first enum member);
- the enum member Green is explicitly given the value 10;
- and the enum member Blue is automatically assigned the value one greater than the member that textually precedes it.

#### end example]

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

[Example: The example

```
enum Circular
{
    A = B,
    B
}
```

results in a compile-time error because the declarations of A and B are circular. A depends on B explicitly, and B depends on A implicitly. *end example*]

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member shall be qualified with the name of its enum type. Enum members do not have any declared accessibility—an enum member is accessible if its containing enum type is accessible.

## 19.5 The System.Enum type

The type System. Enum is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from System. Enum are available in any enum type. A boxing conversion (§11.2.8) exists from any enum type to System. Enum, and an unboxing conversion (§11.3.6) exists from System. Enum to any enum type.

Note that System.Enum is not itself an *enum-type*. Rather, it is a *class-type* from which all *enum-types* are derived. The type System.Enum inherits from the type System.ValueType (§9.3.2), which, in turn, inherits from type object. At run-time, a value of type System.Enum can be null or a reference to a boxed value of any enum type.

## 19.6 Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion (§11.3.3) is required to convert between an enum type and an integral type, or between two enum types. The set of values of the enum type is the same as the set of values of the underlying type and is not restricted to the values of the named constants. Any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type.

Enum members have the type of their containing enum type (except within other enum member initializers: see §19.4). The value of an enum member declared in enum type E with associated value v is (E)v.

The following operators can be used on values of enum types:

- ==, !=, <, >, <=, >= (§12.11.6)
- binary + (§12.9.5)
- binary (§12.9.6)
- ^, &, | (§12.12.3)
- ~ (§12.8.5)
- ++, -- (§12.7.10 and §12.8.6)
- sizeof (§23.6.9)

Every enum type automatically derives from the class System.Enum (which, in turn, derives from System.ValueType and object). Thus, inherited methods and properties of this class can be used on values of an enum type.

# 20. Delegates

# 20.1 General

A delegate declaration defines a class that is derived from the class System.Delegate. A delegate instance encapsulates an *invocation list*, which is a list of one or more methods, each of which is referred to as a *callable entity*. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate's callable entities to be invoked with the given set of arguments.

[*Note*: An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (§20.4) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation. *end note*]

# 20.2 Delegate declarations

A *delegate-declaration* is a *type-declaration* (§14.7) that declares a new delegate type.

delegate-declaration: attributes<sub>opt</sub> delegate-modifiers<sub>opt</sub> delegate return-type identifier variant-type-parameter-list<sub>opt</sub> ( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub> ; delegate modifiers;

delegate-modifiers: delegate-modifier delegate-modifiers delegate-modifier

delegate-modifier: new public protected

protecte internal private

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

A delegate declaration shall not supply a *type-parameter-constraints-clauses* unless it also supplies a *variant-type-parameter-list*.

A delegate declaration that supplies a *variant-type-parameter-list* is a generic delegate declaration.

The new modifier is only permitted on delegates declared within another type, in which case it specifies that such a delegate hides an inherited member by the same name, as described in §15.3.5.

The public, protected, internal, and private modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers might not be permitted (§8.5.2).

The delegate's type name is *identifier*.

The optional *formal-parameter-list* specifies the parameters of the delegate, and *return-type* indicates the return type of the delegate.

The optional variant-type-parameter-list (§18.2.3) specifies the type parameters to the delegate itself.

The return type of a delegate type shall be either void, or output-safe (§18.2.3.2).

All the formal parameter types of a delegate type shall be input-safe. In addition, any out or ref parameter types shall also be output-safe. [Note: Even out parameters are required to be input-safe, due to common implementation restrictions. end note]

Delegate types in C# are name equivalent, not structurally equivalent.

[Example:

delegate int D1(int i, double d); delegate int D2(int c, double d);

The delegate types D1 and D2 are two different types, so they are not interchangeable, despite their identical signatures. *end example*]

Like other generic type declarations, type arguments shall be given to create a constructed delegate type. The parameter types and return type of a constructed delegate type are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type.

The only way to declare a delegate type is via a *delegate-declaration*. Every delegate type is a reference type that is derived from System.Delegate. The members required for every delegate type are detailed in §20.3. Delegate types are implicitly sealed, so it is not permissible to derive any type from a delegate type. It is also not permissible to declare a non-delegate class type deriving from System.Delegate. System.Delegate is not itself a delegate type; it is a class type from which all delegate types are derived.

## 20.3 Delegate members

Every delegate type inherits members from the Delegate class as described in §15.3.4. In addition, every delegate type must provide a non-generic Invoke method whose parameter list matches the *formal-parameter-list* in the delegate declaration, and whose return type matches the *return-type* in the delegate declaration. The Invoke method shall be at least as accessible as the containing delegate type. Calling the Invoke method on a delegate type is semantically equivalent to using the delegate invocation syntax (§20.6).

Implementations may define additional members in the delegate type.

Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the System.Delegate type via the usual member access syntax.

## 20.4 Delegate compatibility

A method or delegate type M is *compatible* with a delegate type D if all of the following are true:

- D and M have the same number of parameters, and each parameter in D has the same ref or out modifiers as the corresponding parameter in M.
- For each value parameter (a parameter with no ref or out modifier), an identity conversion (§11.2.2) or implicit reference conversion (§11.2.7) exists from the parameter type in D to the corresponding parameter type in M.
- For each ref or out parameter, the parameter type in D is the same as the parameter type in M.
- An identity or implicit reference conversion exists from the return type of M to the return type of D.

This definition of consistency allows covariance in return type and contravariance in parameter types.

[Example:

delegate int D1(int i, double d); delegate int D2(int c, double d); delegate object D3(string s);

```
class A
{
    public static int M1(int a, double b) {...}
}
class B
{
    public static int M1(int f, double g) {...}
    public static void M2(int k, double 1) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
    public static object M5(string s) {...}
    public static int[] M6(object o) {...}
}
```

The methods A.M1 and B.M1 are compatible with both the delegate types D1 and D2, since they have the same return type and parameter list. The methods B.M2, B.M3, and B.M4 are incompatible with the delegate types D1 and D2, since they have different return types or parameter lists. The methods B.M5 and B.M6 are both compatible with delegate type D3. *end example*]

[Example:

```
delegate bool Predicate<T>(T value);
class X
{
   static bool F(int i) {...}
   static bool G(string s) {...}
}
```

The method X.F is compatible with the delegate type Predicate<int> and the method X.G is compatible with the delegate type Predicate<string>. *end example*]

[*Note*: The intuitive meaning of delegate compatibility is that a method is compatible with a delegate type if every invocation of the delegate could be replaced with an invocation of the method without violating type safety, treating optional parameters and parameter arrays as explicit parameters. For example, in the following code:

```
delegate void Action<T>(T arg);
class Test {
    static void Print(object value) {
        Console.WriteLine(value);
    }
    static void Main() {
        Action<string> log = Print;
        log("text");
    }
}
```

The Print method is compatible with the Action<string> delegate type because any invocation of an Action<string> delegate would also be a valid invocation of the Print method.

If the signature of the Print method above were changed to Print(object value, bool prependTimestamp = false) for example, the Print method would no longer be compatible with Action<string> by the rules of this clause. *end note*]

# 20.5 Delegate instantiation

An instance of a delegate is created by a *delegate-creation-expression* (§12.7.11.6), a conversion to a delegate type, delegate combination or delegate removal. The newly created delegate instance then refers to one or more of:

- The static method referenced in the *delegate-creation-expression*, or
- The target object (which cannot be null) and instance method referenced in the *delegate-creation-expression*, or
- Another delegate (§12.7.11.6).

[Example:

```
delegate void D(int x);
class C
{
   public static void M1(int i)
                                   {...}
   public void M2(int i)
}
class Test
   static void Main() {
      D cd1 = new D(C.M1);
                               // static method
      C t = new C()
      D cd2 = new D(t.M2);
                                   instance method
      D cd3 = new D(cd2);
                               // another delegate
   }
}
```

#### end example]

The set of methods encapsulated by a delegate instance is called an *invocation list*. When a delegate instance is created from a single method, it encapsulates that method, and its invocation list contains only one entry. However, when two non-null delegate instances are combined, their invocation lists are concatenated—in the order left operand then right operand—to form a new invocation list, which contains two or more entries.

When a new delegate is created from a single delegate the resultant invocation list has just one entry, which is the source delegate (§12.7.11.6).

Delegates are combined using the binary + (\$12.9.5) and += operators (\$12.18.3). A delegate can be removed from a combination of delegates, using the binary – (\$12.9.6) and –= operators (\$12.18.3). Delegates can be compared for equality (\$12.11.9).

[*Example*: The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);
class C
   public static void M1(int i)
   public static void M1(int i) {...}
public static void M2(int i) {...}
}
class Test
   static void Main() {
      D cd1 = new D(C.M1);
                                // M1 - one entry in invocation list
                                // M2 - one entry
      D cd2 = new D(C.M2);
      D cd3 = cd1 + cd2;
                                // M1 + M2 - two entries
      D \ cd4 = cd3 + cd1;
                                // M1 + M2 + M1 - three entries
                                // M1 + M2 + M1 + M1 + M2 - five entries
      D \ cd5 = cd4 + cd3;
      D td3 = new D(cd3);
                                // [M1 + M2] - ONE entry in invocation
                      // list, which is itself a list of two methods.
                                // [M1 + M2] + M1 - two entries
      D td4 = td3 + cd1;
      D cd6 = cd4 - cd2;
                                // M1 + M1 - two entries in invocation list
      D td6 = td4 - cd2;
                                // [M1 + M2] + M1 - two entries in
```

// invocation list, but still three methods called, M2 not removed.
}

When cd1 and cd2 are instantiated, they each encapsulate one method. When cd3 is instantiated, it has an invocation list of two methods, M1 and M2, in that order. cd4's invocation list contains M1, M2, and M1, in that order. For cd5, the invocation list contains M1, M2, M1, M1, and M2, in that order.

When cd1 and cd2 are instantiated, they each encapsulate one method. When cd3 is instantiated, it has an invocation list of two methods, M1 and M2, in that order. cd4's invocation list contains M1, M2, and M1, in that order. For cd5 the invocation list contains M1, M2, M1, M1, and M2, in that order.

When creating a delegate from another delegate with a *delegate-creation-expression* the result has an invocation list with a different structure from the original, but which results in the same methods being invoked in the same order. When td3 is created from cd3 its invocation list has just one member, but that member is a list of the methods M1 and M2 and those methods are invoked by td3 in the same order as they are invoked by cd3. Similarly when td4 is instantiated its invocation list has just two entries but it invokes the three methods M1, M2, and M1, in that order just as cd4 does.

The structure of the invocation list affects delegate subtraction. Delegate cd6, created by subtracting cd2 (which invokes M2) from cd4 (which invokes M1, M2, and M1) invokes M1 and M1. However delegate td6, created by subtracting cd2 (which invokes M2) from td4 (which invokes M1, M2, and M1) still invokes M1, M2 and M1, in that order, as M2 is not a single entry in the list but a member of a nested list.

For more examples of combining (as well as removing) delegates, see §20.6. end example]

Once instantiated, a delegate instance always refers to the same invocation list. [*Note*: Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged. *end note*]

# 20.6 Delegate invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry, is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. (See §12.7.6.4 for detailed information on delegate invocation.) If an exception occurs during the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method to which that delegate referred.

Invocation of a delegate instance whose invocation list contains multiple entries, proceeds by invoking each of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes reference parameters (§15.6.2.3), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list. If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

Attempting to invoke a delegate instance whose value is null results in an exception of type System.NullReferenceException.

[*Example*: The following example shows how to instantiate, combine, remove, and invoke delegates:

using System;

```
delegate void D(int x);
class C
{
   public static void M1(int i) {
    Console.WriteLine("C.M1: " + i);
   }
   public static void M2(int i) {
    Console.WriteLine("C.M2: " + i);
   }
   public void M3(int i) {
    Console.WriteLine("C.M3: " + i);
}
class Test
   static void Main() {
       D cd1 = new D(C.M1);
                       // call M1
       cd1(-1);
       D cd2 = new D(C.M2);
       cd2(-2);
                       // call M2
       D cd3 = cd1 + cd2;
cd3(10); // call M1 then M2
       cd3 += cd1;
       cd3(20);
                       // call M1, M2, then M1
       C c = new C();
       D cd4 = new D(c.M3);
       cd3 += cd4;
cd3(30);
                       // call M1, M2, M1, then M3
       cd3 -= cd1; // remove last M1
cd3(40); // call M1, M2, then M3
        cd3 -= cd4;
       cd3(50);
                       // call M1 then M2
       cd3 -= cd2;
       cd3(60); // call M1
cd3 -= cd2; // impossible removal is benign
       cd3(60);
                       // call M1
        cd3(60);
       cd3 -= cd1; // invocation list is empty so cd3 is null
       cd3(70); // System.NullReferenceException
cd3 -= cd1; // impossible removal is benign
//
                           System.NullReferenceException thrown
   }
}
```

As shown in the statement cd3 += cd1;, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement, cd3 = cd1;, the delegate cd3 refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from a non-empty list) is not an error.

The output produced is:

C.M1:	-1
C.M2:	-2
C.M1:	10
C.M2:	10
C.M1:	20
C.M2:	20
C.M1:	20

C.M1:	30
C.M2:	30
C.M1:	30
C.M3:	30
C.M1:	40
c.M2:	40
C.M3:	40
C.M1:	50
C.M2:	50
C.M1:	60
C.M1:	60

end example]

# 21. Exceptions

# 21.1 General

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application-level error conditions.

# 21.2 Causes of exceptions

Exception can be thrown in two different ways.

- A throw statement (§13.10.6) throws an exception immediately and unconditionally. Control never reaches the statement immediately following the throw.
- Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. [Example: An integer division operation (§12.9.3) throws a System.DivideByZeroException if the denominator is zero. end example] See §21.5 for a list of the various exceptions that can occur in this way.

# 21.3 The System.Exception class

The System.Exception class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- Message is a read-only property of type string that contains a human-readable description of the reason for the exception.
- InnerException is a read-only property of type Exception. If its value is non-null, it refers to the exception that caused the current exception. (That is, the current exception was raised in a catch block handling the InnerException.) Otherwise, its value is null, indicating that this exception was not caused by another exception. The number of exception objects chained together in this manner can be arbitrary.

The value of these properties can be specified in calls to the instance constructor for System. Exception.

## 21.4 How exceptions are handled

Exceptions are handled by a try statement (§13.11).

When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the run-time type of the exception. First, the current method is searched for a lexically enclosing try statement, and the associated catch clauses of the try statement are considered in order. If that fails, the method that called the current method is searched for a lexically enclosing try statement of the call to the current method. This search continues until a catch clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the run-time type of the exception being thrown. A catch clause that doesn't name an exception class can handle any exception.

Once a matching catch clause is found, the system prepares to transfer control to the first statement of the catch clause. Before execution of the catch clause begins, the system first executes, in order, any finally clauses that were associated with try statements more nested that than the one that caught the exception.

If no matching catch clause is found:

- If the search for a matching catch clause reaches a static constructor (§15.12) or static field initializer, then a System.TypeInitializationException is thrown at the point that triggered the invocation of the static constructor. The inner exception of the System.TypeInitializationException contains the exception that was originally thrown.
- Otherwise, if an exception occurs during finalizer execution, and that exception is not caught, then the behavior is unspecified.
- Otherwise, if the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

# 21.5 Common exception classes

The following exceptions are thrown by certain C# operations.

System.ArithmeticException	A base class for exceptions that occur during
	arithmetic operations, such as
	System.DivideByZeroException and
	System.OverflowException.
System.ArrayTypeMismatchException	Thrown when a store into an array fails because the type of the stored element is incompatible with the type of the array.
System.DivideByZeroException	Thrown when an attempt to divide an integral value by zero occurs.
System.IndexOutOfRangeException	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
System.InvalidCastException	Thrown when an explicit conversion from a base type or interface to a derived type fails at run- time.
System.NullReferenceException	Thrown when a null reference is used in a way that causes the referenced object to be required.
System.OutOfMemoryException	Thrown when an attempt to allocate memory (via new) fails.
System.OverflowException	Thrown when an arithmetic operation in a checked context overflows.
System.StackOverflowException	Thrown when the execution stack is exhausted by having too many pending calls; typically indicative of very deep or unbounded recursion.
System.TypeInitializationException	Thrown when a static constructor or static field initializer throws an exception, and no catch clause exists to catch it.

# 22. Attributes

# 22.1 General

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method-modifiers* public, protected, internal, and private.

C# enables programmers to invent new kinds of declarative information, called **attributes**. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. [*Note*: For instance, a framework might define a HelpAttribute attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation. *end note*]

Attributes are defined through the declaration of attribute classes (§22.2), which can have positional and named parameters (§22.2.3). Attributes are attached to entities in a C# program using attribute specifications (§22.3), and can be retrieved at run-time as attribute instances (§22.4).

# 22.2 Attribute classes

### 22.2.1 General

A class that derives from the abstract class System.Attribute, whether directly or indirectly, is an *attribute class*. The declaration of an attribute class defines a new kind of attribute that can be placed on program entities. By convention, attribute classes are named with a suffix of Attribute. Uses of an attribute may either include or omit this suffix.

A generic class declaration shall not use System. Attribute as a direct or indirect base class. [Example:

```
using System;
public class B : Attribute {}
public class C<T> : B {} // Error - generic cannot be an attribute
```

end example]

## 22.2.2 Attribute usage

The attribute AttributeUsage (§22.5.2) is used to describe how an attribute class can be used.

AttributeUsage has a positional parameter (§22.2.3) that enables an attribute class to specify the kinds of program entities on which it can be used. [*Example*: The example

```
using System;
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{ ... }
```

defines an attribute class named SimpleAttribute that can be placed on *class-declarations* and *interface-declarations* only. The example

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

shows several uses of the Simple attribute. Although this attribute is defined with the name SimpleAttribute, when this attribute is used, the Attribute suffix may be omitted, resulting in the short name Simple. Thus, the example above is semantically equivalent to the following

```
[SimpleAttribute] class Class1 {...}
[SimpleAttribute] interface Interface1 {...}
```

#### end example]

AttributeUsage has a named parameter (§22.2.3), called AllowMultiple, which indicates whether the attribute can be specified more than once for a given entity. If AllowMultiple for an attribute class is true, then that attribute class is a *multi-use attribute class*, and can be specified more than once on an entity. If AllowMultiple for an attribute class is false or it is unspecified, then that attribute class is a *single-use attribute class*, and can be specified at most once on an entity.

[Example: The example

```
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;
    public AuthorAttribute(string name) {
       this.name = name;
    }
    public string Name {
       get { return name; }
    }
}
```

defines a multi-use attribute class named AuthorAttribute. The example

[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1 {...}

shows a class declaration with two uses of the Author attribute. end example]

AttributeUsage has another named parameter (§22.2.3), called Inherited, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If Inherited for an attribute class is true, then that attribute is inherited. If Inherited for an attribute class is false then that attribute is not inherited. If is unspecified, its default value is true.

An attribute class X not having an AttributeUsage attribute attached to it, as in

```
using System;
class X: Attribute { ... }
```

is equivalent to the following:

```
using System;
[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute { ... }
```

#### 22.2.3 Positional and named parameters

Attribute classes can have **positional parameters** and **named parameters**. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class. Both accessors of a property need to be public for the property to define a named parameter.

[Example: The example

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) { // url is a positional parameter
    }
    public string Topic { // Topic is a named parameter
        get {...}
        set {...}
    }
    public string Url { get {...} }
}
```

defines an attribute class named HelpAttribute that has one positional parameter, url, and one named parameter, Topic. Although it is non-static and public, the property Url does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class class1
{
}
[Help("http://www.mycompany.com/.../Misc.htm", Topic ="Class2")]
class Class2
{
}
```

end example]

#### 22.2.4 Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the *attribute parameter types*, which are:

- One of the following types: bool, byte, char, double, float, int, long, sbyte, short, string, uint, ulong, ushort.
- The type object.
- The type System.Type.
- Enum types.
- Single-dimensional arrays of the above types.

A constructor argument or public field that does not have one of these types, shall not be used as a positional or named parameter in an attribute specification.

# 22.3 Attribute specification

Attribute specification is the application of a previously defined attribute to a program entity. An attribute is a piece of additional declarative information that is specified for a program entity. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for *type-declarations* (§14.7), *class-member-declarations* (§15.3), *interface-member-declarations* (§18.4), *struct-member-declarations* (§16.3), *enum-member-declarations* (§19.2), *accessor-declarations* (§15.7.3), *event-accessor-declarations* (§15.8), elements of *formal-parameter-lists* (§15.6.2), and elements of *type-parameter-lists* (§15.2.3).

Attributes are specified in *attribute sections*. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications [A] [B], [B] [A], [A, B], and [B, A] are equivalent.

```
global-attributes:
    global-attribute-sections
alobal-attribute-sections:
    global-attribute-section
    global-attribute-sections global-attribute-section
alobal-attribute-section:
    [ qlobal-attribute-target-specifier attribute-list ]
    [ global-attribute-target-specifier attribute-list , ]
global-attribute-target-specifier:
    global-attribute-target :
qlobal-attribute-target:
    identifier equal to assembly or module
attributes:
    attribute-sections
attribute-sections:
    attribute-section
    attribute-sections attribute-section
attribute-section:
    [ attribute-target-specifier<sub>opt</sub> attribute-list ]
    [ attribute-target-specifier<sub>opt</sub> attribute-list , ]
attribute-target-specifier:
    attribute-target :
attribute-target:
    identifier not equal to assembly or module
    keyword
attribute-list:
    attribute
    attribute-list, attribute
attribute:
    attribute-name attribute-arguments<sub>opt</sub>
attribute-name:
    type-name
```

attribute-arguments: ( *positional-argument-list*<sub>opt</sub> ) ( *positional-argument-list* , *named-argument-list* ) ( *named-argument-list* ) positional-araument-list: positional-argument positional-argument-list , positional-argument positional-argument: argument-name<sub>opt</sub> attribute-argument-expression named-argument-list: named-araument named-argument-list , named-argument named-argument: *identifier* = *attribute-argument-expression* attribute-argument-expression: expression

For the above productions *global-attribute-target* and *attribute-target*, and in the text below, the referenced equality is that defined in §7.4.3.

An attribute consists of an *attribute-name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute-argument-expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute-argument-expression*, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

[*Note*: For convenience, a trailing comma is allowed in a *global-attribute-section* and an *attribute-section*, just as one is allowed in an *array-initializer* (§17.7). *end note*]

The attribute-name identifies an attribute class.

When an attribute is placed at the global level, a *global-attribute-target-specifier* is required. When the *global-attribute-target* is equal to:

- assembly the target is the containing assembly
- module the target is the containing module

No other values for global-attribute-target are allowed.

The standardized *attribute-target* names are event, field, method, param, property, return, type, and typevar. These target names shall only be used in the following contexts:

- event an event.
- field a field. A field-like event (i.e., one without accessors) can also have an attribute with this target.
- method a constructor, finalizer, method, operator, property get and set accessors, indexer get and set
  accessors, and event add and remove accessors. A field-like event (i.e., one without accessors) can also
  have an attribute with this target.
- param a property set accessor, an indexer set accessor, event add and remove accessors, and a
  parameter in a constructor, method, and operator.
- property a property and an indexer.
- return a delegate, method, operator, property get accessor, and indexer get accessor.

- type a delegate, class, struct, enum, and interface.
- typevar a type parameter.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute-target-specifier*. Without an *attribute-target-specifier* a default is applied, but an *attribute-target-specifier* can be used to affirm or override the default. The contexts are resolved as follows:

- For an attribute on a delegate declaration the default target is the delegate. Otherwise when the attribute-target is equal to:
  - type the target is the delegate
  - return the target is the return value
- For an attribute on a method declaration the default target is the method. Otherwise when the *attribute-target* is equal to:
  - method the target is the method
  - o return the target is the return value
- For an attribute on an operator declaration the default target is the operator. Otherwise when the *attribute-target* is equal to:
  - method the target is the operator
  - o return the target is the return value
- For an attribute on a get accessor declaration for a property or indexer declaration the default target is the associated method. Otherwise when the *attribute-target* is equal to:
  - method the target is the associated method
  - o return the target is the return value
- For an attribute specified on a set accessor for a property or indexer declaration the default target is the associated method. Otherwise when the *attribute-target* is equal to:
  - method the target is the associated method
  - o param the target is the lone implicit parameter
- For an attribute specified on an event declaration that omits event-accessor-declarations the default target is the event declaration. Otherwise when the *attribute-target* is equal to:
  - event the target is the event declaration
  - o field the target is the field
  - $\circ$  method the targets are the methods
- In the case of an event declaration that does not omit event-accessor-declarations the default target is the method.
  - method the target is the associated method
  - o param the target is the lone parameter

In all other contexts, inclusion of an *attribute-target-specifier* is permitted but unnecessary. [*Example*: a class declaration may either include or omit the specifier type:

```
[type: Author("Brian Kernighan")]
class Class1 {}
[Author("Dennis Ritchie")]
class Class2 {}
```

end example.]

An implementation can accept other *attribute-targets*, the purposes of which are implementation defined. An implementation that does not recognize such an *attribute-target* shall issue a warning and ignore the containing *attribute-section*.

By convention, attribute classes are named with a suffix of Attribute. An *attribute-name* can either include or omit this suffix. Specifically, an *attribute-name* is resolved as follows:

- If the right-most identifier of the *attribute-name* is a verbatim identifier (§7.4.3), then the *attribute-name* is resolved as a *type-name* (§8.8). If the result is not a type derived from System.Attribute, a compile-time error occurs.
- Otherwise,
  - The *attribute-name* is resolved as a *type-name* (§8.8) except any errors are suppressed. If this resolution is successful and results in a type derived from System.Attribute then the type is the result of this step.
  - The characters Attribute are appended to the right-most identifier in the attribute-name and the resulting string of tokens is resolved as a type-name (§8.8) except any errors are suppressed. If this resolution is successful and results in a type derived from System.Attribute then the type is the result of this step.

If exactly one of the two steps above results in a type derived from System.Attribute, then that type is the result of the *attribute-name*. Otherwise a compile-time error occurs.

[*Example*: If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the *attribute-name* is spelled such that its right-most *identifier* is a verbatim identifier (§7.4.3), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class Example: Attribute
{}
[AttributeUsage(AttributeTargets.A]])]
public class ExampleAttribute: Attribute
[Example]
                        // Error: ambiguity
class Class1 {}
[ExampleAttribute]
                        // Refers to ExampleAttribute
class Class2 {}
[@Example]
                        // Refers to Example
class class3 {}
[@ExampleAttribute]
                        // Refers to ExampleAttribute
class Class4 {}
```

shows two attribute classes named Example and ExampleAttribute. The attribute [Example] is ambiguous, since it could refer to either Example or ExampleAttribute. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute [ExampleAttribute] is not ambiguous

(although it would be if there was an attribute class named ExampleAttributeAttribute!). If the declaration for class Example is removed, then both attributes refer to the attribute class named ExampleAttribute, as follows:

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class ExampleAttribute: Attribute
{}
[Example] // Refers to ExampleAttribute
class Class1 {}
[ExampleAttribute] // Refers to ExampleAttribute
class Class2 {}
[@Example] // Error: no attribute named "Example"
class Class3 {}
```

end example]

It is a compile-time error to use a single-use attribute class more than once on the same entity. [*Example*: The example

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;
    public HelpStringAttribute(string value) {
       this.value = value;
    }
    public string Value { get {...} }
}
[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use HelpString, which is a single-use attribute class, more than once on the declaration of Class1. *end example*]

An expression E is an *attribute-argument-expression* if all of the following statements are true:

- The type of E is an attribute parameter type (§22.2.4).
- At compile-time, the value of E can be resolved to one of the following:
  - o A constant value.
  - A System. Type object obtained using a *typeof-expression* (§12.7.12) specifying a non-generic type, a closed constructed type (§9.4.3), or an unbound generic type (§9.4.4), but not an open type (§9.4.3).
  - A single-dimensional array of *attribute-argument-expressions*.

[Example:

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }
}
```

```
public Type P2 {
      get {...}
set {...}
   }
   public object P3 {
      get {...}
set {...}
   }
}
[Test(P1 = 1234, P3 = new int[]{1, 3, 5}, P2 = typeof(float))]
class MyClass {}
class C<T> {
   [My(P2 = typeof(T))]
                                // Error - T not a closed type.
   int x1;
   [My(P2 = typeof(C < T >))]
                                 // Error - C<T> not a closed type.
   int x2;
   [My(P2 = typeof(C < int >))] // Ok
   int x3:
   [My(P2 = typeof(C<>))]
                                 // Ok
   int x4;
}
```

#### end example]

The attributes of a type declared in multiple parts are determined by combining, in an unspecified order, the attributes of each of its parts. If the same attribute is placed on multiple parts, it is equivalent to specifying that attribute multiple times on the type. [*Example*: The two parts:

```
[Attr1, Attr2("hello")]
partial class A {}
[Attr3, Attr2("goodbye")]
partial class A {}
```

are equivalent to the following single declaration:

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

end example]

Attributes on type parameters combine in the same way.

## 22.4 Attribute instances

#### 22.4.1 General

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 subclauses.

#### 22.4.2 Compilation of an attribute

The compilation of an *attribute* with attribute class T, *positional-argument-list* P, *named-argument-list* N, and specified on a program entity E is compiled into an assembly A viathe following steps:

- Follow the compile-time processing steps for compiling an *object-creation-expression* of the form new T(P). These steps either result in a compile-time error, or determine an instance constructor C on T that can be invoked at run-time.
- If C does not have public accessibility, then a compile-time error occurs.
- For each *named-argument* Arg in N:
  - Let Name be the *identifier* of the *named-argument* Arg.
  - Name shall identify a non-static read-write public field or property on T. If T has no such field or property, then a compile-time error occurs.
- If any of the values within *positional-argument-list* P or one of the values within *named-argument-list* N is of type System.String and the value is not well-formed as defined by the Unicode Standard, it is implementation-defined whether the value compiled is equal to the run-time value retrieved (§22.4.3). [*Note:* As an example, a string which contains a high surrogate UTF-16 code unit which isn't immediately followed by a low surrogate code unit is not well-formed. *end note*]
- Store the following information (for run-time instantiation of the attribute) in the assembly output by the compiler as a result of compiling the program containing the attribute: the attribute class T, the instance constructor C on T, the *positional-argument-list* P, the *named-argument-list* N, and the associated program entity E, with the values resolved completely at compile-time.

#### 22.4.3 Run-time retrieval of an attribute instance

The attribute instance represented by T, C, P, and N, and associated with E can be retrieved at run-time from the assembly A using the following steps:

- Follow the run-time processing steps for executing an *object-creation-expression* of the form new T(P), using the instance constructor C and values as determined at compile-time. These steps either result in an exception, or produce an instance O of T.
- For each *named-argument* Arg in N, in order:
  - Let Name be the *identifier* of the *named-argument* Arg. If Name does not identify a non-static public read-write field or property on O, then an exception is thrown.
  - Let Value be the result of evaluating the *attribute-argument-expression* of Arg.
  - $\circ$  If Name identifies a field on O, then set this field to Value.
  - Otherwise, Name identifies a property on O. Set this property to Value.
  - The result is O, an instance of the attribute class T that has been initialized with the *positional*argument-list P and the *named*-argument-list N.

[*Note*: The format for storing T, C, P, N (and associating it with E) in A and the mechanism to specify E and retrieve T, C, P, N from A (and hence how an attribute instance is obtained at runtime) is beyond the scope of this standard. *end note*]

[*Example*: In an implementation of the CLI, the Help attribute instances in the assembly created by compiling the example program in §22.2.3 can be retrieved with the following program:

end example]

# 22.5 Reserved attributes

#### 22.5.1 General

A small number of attributes affect the language in some way. These attributes include:

- System.AttributeUsageAttribute (§22.5.2), which is used to describe the ways in which an attribute class can be used.
- System.Diagnostics.ConditionalAttribute (§22.5.3), is a multi-use attribute class which is used to define conditional methods and conditional attribute classes. This attribute indicates a condition by testing a conditional compilation symbol.
- System.ObsoleteAttribute (§22.5.4), which is used to mark a member as obsolete.
- System.Runtime.CompilerServices.CallerLineNumberAttribute (§22.5.5.2), System.Runtime.CompilerServices.CallerFilePathAttribute (§22.5.5.3), and System.Runtime.CompilerServices.CallerMemberNameAttribute (§22.5.5.4), which are used to supply information about the calling context to optional parameters.

An execution environment may provide additional implementation-specific attributes that affect the execution of a C# program.

#### 22.5.2 The AttributeUsage attribute

The attribute AttributeUsage is used to describe the manner in which the attribute class can be used.

A class that is decorated with the AttributeUsage attribute shall derive from System.Attribute, either directly or indirectly. Otherwise, a compile-time error occurs.

[Note: For an example of using this attribute, see §22.2.2. end note]

#### 22.5.3 The Conditional attribute

22.5.3.1 General

The attribute Conditional enables the definition of *conditional methods* and *conditional attribute classes*.

#### 22.5.3.2 Conditional methods

A method decorated with the Conditional attribute is a conditional method. Each conditional method is thus associated with the conditional compilation symbols declared in its Conditional attributes. [*Example*:

```
using System.Diagnostics;
class Eg
{
    [Conditional("ALPHA")]
    [Conditional("BETA")]
    public static void M() {
        //...
    }
}
```

declares Eg.M as a conditional method associated with the two conditional compilation symbols ALPHA and BETA. *end example*]

A call to a conditional method is included if one or more of its associated conditional compilation symbols is defined at the point of call, otherwise the call is omitted.

A conditional method is subject to the following restrictions:

- The conditional method shall be a method in a *class-declaration* or *struct-declaration*. A compile-time error occurs if the Conditional attribute is specified on a method in an interface declaration.
- The conditional method shall have a return type of void.
- The conditional method shall not be marked with the override modifier. A conditional method can be marked with the virtual modifier, however. Overrides of such a method are implicitly conditional, and shall not be explicitly marked with a Conditional attribute.
- The conditional method shall not be an implementation of an interface method. Otherwise, a compile-time error occurs.
- The parameters of the conditional method shall not have the out modifier.

In addition, a compile-time error occurs if a delegate is created from a conditional method.

#### [Example: The example

```
#define DEBUG
using System;
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}
class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

declares Class1.M as a conditional method. Class2's Test method calls this method. Since the conditional compilation symbol DEBUG is defined, if Class2.Test is called, it will call M. If the symbol DEBUG had not been defined, then Class2.Test would not call Class1.M. *end example*]

It is important to understand 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 following code

```
// File class1.cs
using System.Diagnostics;
```

```
class Class1
{
   [Conditional("DEBUG")]
   public static void F() {
    Console.WriteLine("Executed Class1.F");
   }
}
// File class2.cs
#define DEBUG
class Class2
Ł
   public static void G() {
                               // F is called
      Class1.F();
}
// File class3.cs
#undef DEBUG
class Class3
{
   public static void H() {
                               // F is not called
      Class1.F();
   }
}
```

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*]

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 following code

```
// File class1.cs
using System;
using System.Diagnostics;
class Class1
   [Conditional("DEBUG")]
   public virtual void M() {
    Console.WriteLine("Class1.M executed");
   }
}
// File class2.cs
using System;
class Class2: Class1{
   public override void M() {
       Console.WriteLine("Class2.M executed");
                                     // base M is not called!
      base.M();
   }
}
```

```
// File class3.cs
#define DEBUG
using System;
class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M(); // M is called
    }
}
```

Class2 includes a call to the M defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol DEBUG, which is undefined. Thus, the method writes to the console "Class2.M executed" only. Judicious use of *pp-declarations* can eliminate such problems. *end example*]

#### 22.5.3.3 Conditional attribute classes

An attribute class (§22.2) decorated with one or more Conditional attributes is a *conditional attribute class*. A conditional attribute class is thus associated with the conditional compilation symbols declared in its Conditional attributes.

[Example:

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declares TestAttribute as a conditional attribute class associated with the conditional compilations symbols ALPHA and BETA. *end example*]

Attribute specifications (§22.3) of a conditional attribute are included if one or more of its associated conditional compilation symbols is defined at the point of specification, otherwise the attribute specification is omitted.

It is important to note that the inclusion or exclusion of an attribute specification of a conditional attribute class is controlled by the conditional compilation symbols at the point of the specification. [*Example*: In the example

```
File test.cs:
    using System;
    using System.Diagnostics;
    [Conditional("DEBUG")]
    public class TestAttribute : Attribute {}
File class1.cs:
    #define DEBUG
    [Test] // TestAttribute is specified
    class Class1 {}
File class2.cs:
    #undef DEBUG
    [Test] // TestAttribute is not specified
```

class Class2 {}

the classes Class1 and Class2 are each decorated with attribute Test, which is conditional based on whether or not DEBUG is defined. Since this symbol is defined in the context of Class1 but not Class2, the specification of the Test attribute on Class1 is included, while the specification of the Test attribute on Class2 is omitted. *end example*]

### 22.5.4 The Obsolete attribute

The attribute Obsolete is used to mark types and members of types that should no longer be used.

If a program uses a type or member that is decorated with the Obsolete attribute, the compiler shall issue a warning or an error. Specifically, the compiler shall issue a warning if no error parameter is provided, or if the error parameter is provided and has the value false. The compiler shall issue an error if the error parameter is specified and has the value false.

[Example: In the following code

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
   public void F() {}
}
class B
{
   public void F() {}
}
class Test
{
   static void Main() {
      A a = new A(); // Warning
      a.F();
   }
}
```

the class A is decorated with the Obsolete attribute. Each use of A in Main results in a warning that includes the specified message, "This class is obsolete; use class B instead." *end example*]

#### 22.5.5 Caller-info attributes

#### 22.5.5.1 General

For purposes such as logging and reporting, it is sometimes useful for a function member to obtain certain compile-time information about the calling code. The caller-info attributes provide a way to pass such information transparently.

When an optional parameter is annotated with one of the caller-info attributes, omitting the corresponding argument in a call does not necessarily cause the default parameter value to be substituted. Instead, if the specified information about the calling context is available, that information will be passed as the argument value.

[Example:

using System.Runtime.CompilerServices

...

```
public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath] string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line "+ line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}</pre>
```

A call to Log() with no arguments would print the line number and file path of the call, as well as the name of the member within which the call occurred. *end example*]

Caller-info attributes can occur on optional parameters anywhere, including in delegate declarations. However, the specific caller-info attributes have restrictions on the types of the parameters they can attribute, so that there will always be an implicit conversion from a substituted value to the parameter type.

It is an error to have the same caller-info attribute on a parameter of both the defining and implementing part of a partial method declaration. Only caller-info attributes in the defining part are applied, whereas caller-info attributes occurring only in the implementing part are ignored.

Caller information does not affect overload resolution. As the attributed optional parameters are still omitted from the source code of the caller, overload resolution ignores those parameters in the same way it ignores other omitted optional parameters (§12.6.4).

Caller information is only substituted when a function is explicitly invoked in source code. Implicit invocations such as implicit parent constructor calls do not have a source location and will not substitute caller information. Also, calls that are dynamically bound will not substitute caller information. When a caller-info attributed parameter is omitted in such cases, the specified default value of the parameter is used instead.

One exception is query expressions. These are considered syntactic expansions, and if the calls they expand to omit optional parameters with caller-info attributes, caller information will be substituted. The location used is the location of the query clause which the call was generated from.

If more than one caller-info attribute is specified on a given parameter, they are preferred in the following order: CallerLineNumber, CallerFilePath, CallerMemberName.

#### 22.5.5.2 The CallerLineNumber attribute

The System.Runtime.CompilerServices.CallerLineNumberAttribute is allowed on optional parameters when there is a standard implicit conversion (§11.2.2) from the constant value int.MaxValue to the parameter's type. This ensures that any non-negative line number up to that value can be passed without error.

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsageAttribute(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerLineNumberAttribute : Attribute
    {
        public CallerLineNumberAttribute() {...}
    }
}
```

If a function invocation from a location in source code omits an optional parameter with the CallerLineNumberAttribute, then a numeric literal representing that location's line number is used as an argument to the invocation instead of the default parameter value.

If the invocation spans multiple lines, the line chosen is implementation-dependent.

The line number may be affected by #line directives (§7.5.8).

22.5.5.3 The CallerFilePath attribute

The System.Runtime.CompilerServices.CallerFilePathAttribute is allowed on optional parameters when there is a standard implicit conversion (§11.2.2) from string to the parameter's type.

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsageAttribute(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerFilePathAttribute : Attribute
    {
        public CallerFilePathAttribute() {...}
    }
}
```

If a function invocation from a location in source code omits an optional parameter with the CallerFilePathAttribute, then a string literal representing that location's file path is used as an argument to the invocation instead of the default parameter value.

The format of the file path is implementation-dependent.

The file path may be affected by #line directives (§7.5.8).

#### 22.5.5.4 The CallerMemberName attribute

The System.Runtime.CompilerServices.CallerMemberNameAttribute is allowed on optional parameters when there is a standard implicit conversion (§11.2.2) from string to the parameter's type.

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsageAttribute(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerMemberNameAttribute : Attribute
    {
        public CallerMemberNameAttribute() {...}
    }
}
```

If a function invocation from a location within the body of a function member or within an attribute applied to the function member itself or its return type, parameters or type parameters in source code omits an optional parameter with the CallerMemberNameAttribute, then a string literal representing the name of that member is used as an argument to the invocation instead of the default parameter value.

For invocations that occur within generic methods, only the method name itself is used, without the type parameter list.

For invocations that occur within explicit interface member implementations, only the method name itself is used, without the preceding interface qualification.

For invocations that occur within property or event accessors, the member name used is that of the property or event itself.

For invocations that occur within indexer accessors, the member name used is that supplied by an IndexerNameAttribute (§) on the indexer member, if present, or the default name Item otherwise.

For invocations that occur within field or event initializers, the member name used is the name of the field or event being initialized.

For invocations that occur within declarations of instance constructors, static constructors, finalizers and operators the member name used is implementation-dependent.

## 22.6 Attributes for interoperation

For interoperation with other languages, an indexer may be implemented using indexed properties. If no IndexerName attribute is present for an indexer, then the name Item is used by default. The IndexerName attribute enables a developer to override this default and specify a different name.

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

# 23. Unsafe code

# 23.1 General

An implementation that does not support unsafe code is required to diagnose any usage of the keyword unsafe.

#### The remainder of this clause, including all of its subclauses, is conditionally normative.

[*Note*: The core C# language, as defined in the preceding clauses, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language, it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm might not be possible or practical without access to pointers. To address this need, C# provides the ability to write **unsafe code**.

In unsafe code, it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is in fact a "safe" feature from the perspective of both developers and users. Unsafe code shall be clearly marked with the modifier unsafe, so developers can't possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment. *end note*]

## 23.2 Unsafe contexts

The unsafe features of C# are available only in unsafe contexts. An unsafe context is introduced by including an unsafe modifier in the declaration of a type or member, or by employing an unsafe statement:

- A declaration of a class, struct, interface, or delegate may include an unsafe modifier, in which case, the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context. [*Note*: If the *type-declaration* is partial, only that part is an unsafe context. *end note*]
- A declaration of a field, method, property, event, indexer, operator, instance constructor, finalizer, or static constructor may include an unsafe modifier, in which case, the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe-statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar extensions are shown below. For brevity, ellipses (...) are used to represent productions that appear in preceding clauses.

*class-modifier:* ... unsafe

```
struct-modifier:
            ...
            unsafe
        interface-modifier:
            ...
            unsafe
        delegate-modifier:
            ...
            unsafe
        field-modifier:
            ...
            unsafe
        method-modifier:
            ...
            unsafe
        property-modifier:
            ...
            unsafe
        event-modifier:
            ...
            unsafe
        indexer-modifier:
            ...
            unsafe
        operator-modifier:
            ...
            unsafe
        constructor-modifier:
            ...
            unsafe
        finalizer-declaration:
            attributes_{opt} extern<sub>opt</sub> unsafe<sub>opt</sub> ~ identifier ( ) finalizer-body
            attributes_{opt} unsafe<sub>opt</sub> extern<sub>opt</sub> ~ identifier ( ) finalizer-body
        static-constructor-modifiers:
            extern<sub>opt</sub> unsafe<sub>opt</sub> static
            unsafeopt externopt static
            externopt static unsafeopt
            unsafe<sub>opt</sub> static extern<sub>opt</sub>
            static externopt unsafeopt
            static unsafeopt externopt
        embedded-statement:
            ...
            unsafe-statement
        unsafe-statement:
            unsafe block
[Example: In the following code
```

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

the unsafe modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the Left and Right fields to be of a pointer type. The example above could also be written

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Here, the unsafe modifiers in the field declarations cause those declarations to be considered unsafe contexts. *end example*]

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 following code

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
    public class B: A
{
    public override void F() {
        base.F();
    }
    ...
}
```

the unsafe modifier on the F method in A simply causes the textual extent of F to become an unsafe context in which the unsafe features of the language can be used. In the override of F in B, there is no need to re-specify the unsafe modifier—unless, of course, the F method in B itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}
public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

Here, because F's signature includes a pointer type, it can 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*]

When the unsafe modifier is used on a partial type declaration (§15.2.7), only that particular part is considered an unsafe context.

# 23.3 Pointer types

In an unsafe context, a *type* (§9) can be a *pointer-type* as well as a *value-type*, a *reference-type*, or a *type-parameter*. In an unsafe context a pointer-type may also be the element type of an array (§17). A pointer-

type may also be used in a typeof expression (§12.7.12) outside of an unsafe context (as such usage is not unsafe).

type: ... pointer-type non-array-type: ... pointer-type

A *pointer-type* is written as an *unmanaged-type* or the keyword void, followed by a \* token:

```
pointer-type:
unmanaged-type *
void *
unmanaged-type:
type
```

The type specified before the \* in a pointer type is called the *referent type* of the pointer type. It represents the type of the variable to which a value of the pointer type points.

A *pointer-type* may only be used in an *array-type* in an unsafe context (§23.2). A *non-array-type* is any type that is not itself an *array-type*.

The type specified before the \* in a pointer type is called the *referent type* of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector—the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer shall be an *unmanaged-type*.

An *unmanaged-type* is any type that isn't a *reference-type*, a type-parameter, or a constructed type, and contains no fields whose type is not an unmanaged-type. In other words, an *unmanaged-type* is one of the following:

- sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool.
- Any enum-type.
- Any pointer-type.
- Any user-defined *struct-type* that is not a constructed type and contains fields of *unmanaged-types* only.

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

[*Example*: Some examples of pointer types are given in the table below:

Example	Description
byte*	Pointer to byte
char*	Pointer to char
int**	Pointer to pointer to int
int*[]	Single-dimensional array of pointers to int
void*	Pointer to unknown type

end example]

For a given implementation, all pointer types shall have the same size and representation.

[*Note*: Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the \* is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example:

int\* pi, pj; // NOT as int \*pi, \*pj;

end note]

The value of a pointer having type T\* represents the address of a variable of type T. The pointer indirection operator \* (§23.6.2) can be used to access this variable. [*Example*: Given a variable P of type int\*, the expression \*P denotes the int variable found at the address contained in P. *end example*]

Like an object reference, a pointer may be null. Applying the indirection operator to a null pointer results in implementation-defined behavior (§23.6.2). A pointer with value null is represented by all-bits-zero.

The void\* type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type void\*, nor can any arithmetic be performed on such a pointer. However, a pointer of type void\* can be cast to any other pointer type (and vice versa) and compared to values of other pointer types (§23.6.8).

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from object and no conversions exist between pointer types and object. In particular, boxing and unboxing (§9.3.12) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in §23.5.

A *pointer-type* cannot be used as a type argument (§9.4), and type inference (§12.6.3) fails on generic method calls that would have inferred a type argument to be a pointer type.

A pointer-type cannot be used as a type of a subexpression of a dynamically bound operation (§12.3.3).

A pointer-type may be used as the type of a volatile field (§15.5.4).

[*Note*: Although pointers can be passed as ref or out parameters, doing so can cause undefined behavior, since the pointer might well be set to point to a local variable that no longer exists when the called method returns, or the fixed object to which it used to point, is no longer fixed. For example:

```
using System;
       class Test
       {
           static int value = 20;
           unsafe static void F(out int* pi1, ref int* pi2) {
               int i = 10;
               pi1 = &i;
               fixed (int* pj = &value) {
                  // ...
pi2 = pj;
               }
           }
           static void Main() {
               int i = 10;
              unsafe {
int* px1;
int* px2 = &i;
                  F(out px1, ref px2);
                  Console.WriteLine("*px1 = {0}, *px2 = {1}",
 *px1, *px2); // undefined behavior
               }
           }
       }
end note]
```

A method can return a value of some type, and that type can be a pointer. [*Example*: When given a pointer to a contiguous sequence of ints, that sequence's element count, and some other int value, the following method returns the address of that value in that sequence, if a match occurs; otherwise it returns null:

```
unsafe static int* Find(int* pi, int size, int value) {
   for (int i = 0; i < size; ++i) {
      if (*pi == value)
        return pi;
      ++pi;
   }
   return null;
}</pre>
```

## end example]

In an unsafe context, several constructs are available for operating on pointers:

- The unary \* operator may be used to perform pointer indirection (§23.6.2).
- The -> operator may be used to access a member of a struct through a pointer (§23.6.3).
- The [] operator may be used to index a pointer (§23.6.4).
- The unary & operator may be used to obtain the address of a variable (§23.6.5).
- The ++ and -- operators may be used to increment and decrement pointers (§23.6.6).
- The binary + and operators may be used to perform pointer arithmetic (§23.6.7).
- The ==, !=, <, >, <=, and >= operators may be used to compare pointers (§23.6.8).
- The stackalloc operator may be used to allocate memory from the call stack (§23.9).
- The fixed statement may be used to temporarily fix a variable so its address can be obtained (§23.7).

## 23.4 Fixed and moveable variables

The address-of operator (§23.6.5) and the fixed statement (§23.7) divide variables into two categories: *Fixed variables* and *moveable variables*.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

The & operator (§23.6.5) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can only be obtained using a fixed statement (§23.7), and that address remains valid only for the duration of that fixed statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple-name* (§12.7.3) that refers to a local variable, value parameter, or parameter array, unless the variable is captured by an anonymous function (§12.16.6.2).
- A variable resulting from a *member-access* (§12.7.5) of the form V.I, where V is a fixed variable of a *struct-type*.
- A variable resulting from a *pointer-indirection-expression* (§23.6.2) of the form \*P, a *pointer-member-access* (§23.6.3) of the form P->I, or a *pointer-element-access* (§23.6.4) of the form P[E].

All other variables are classified as moveable variables.

A static field is classified as a moveable variable. Also, a ref or out parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, a variable produced by dereferencing a pointer is always classified as a fixed variable.

# 23.5 Pointer conversions

## 23.5.1 General

In an unsafe context, the set of available implicit conversions (§11.2) is extended to include the following implicit pointer conversions:

- From any *pointer-type* to the type void\*.
- From the null literal (§7.4.5.7) to any *pointer-type*.

Additionally, in an unsafe context, the set of available explicit conversions (§11.3) is extended to include the following explicit pointer conversions:

- From any *pointer-type* to any other *pointer-type*.
- From sbyte, byte, short, ushort, int, uint, long, or ulong to any *pointer-type*.
- From any *pointer-type* to sbyte, byte, short, ushort, int, uint, long, or ulong.

Finally, in an unsafe context, the set of standard implicit conversions (§11.4.2) includes the following pointer conversions:

- From any *pointer-type* to the type void\*.
- From the null literal to any *pointer-type*.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which, in turn, is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C. [*Example*: Consider the following case in which a variable having one type is accessed via a pointer to a different type:

## end example]

When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. [*Example*: The following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;
class Test
{
    static void Main() {
        double d = 123.456e23;
        unsafe {
            byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
                Console.Write("{0:x2} ", *pb++);
               Console.WriteLine();
        }
    }
}
```

Of course, the output produced depends on endianness. end example]

Mappings between pointers and integers are implementation-defined. [*Note*: However, on 32- and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of uint or ulong values, respectively, to or from those integral types. *end note*]

## 23.5.2 Pointer arrays

Arrays of pointers can be constructed using *array-creation-expression* (§12.7.11.5) in an usafe context. Only some of the conversions that apply to other array types are allowed on pointer arrays:

- The implicit reference conversion (§11.2.5) from any *array-type* to System.Array and the interfaces it implements also applies to pointer arrays. However, any attempt to access the array elements through System.Array or the interfaces it implements may result in an exception at run-time, as pointer types are not convertible to object.
- The implicit and explicit reference conversions (§11.2.5, §11.3.4) from a single-dimensional array type S[] to System.Collections.Generic.IList<T> and its generic base interfaces never apply to pointer arrays.
- The explicit reference conversion (§11.3.4) from System.Array and the interfaces it implements to any *array-type* applies to pointer arrays.
- The explicit reference conversions (§11.3.4) from System.Collections.Generic.IList<S> and its base interfaces to a single-dimensional array type T[] never applies to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.

These restrictions mean that the expansion for the foreach statement over arrays described in §10.4.4.17 cannot be applied to pointer arrays. Instead, a foreach statement of the form

foreach (V v in x) embedded-statement

where the type of x is an array type of the form T[, ,..., ], *n* is the number of dimensions minus 1 and T or V is a pointer type, is expanded using nested for-loops as follows:

```
{
    T[,,...,] a = x; for (int i0 = a.GetLowerBound(0); i0 <=
a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {
        V v = (V)a[i0,i1,...,in];
        embedded-statement
    }
}</pre>
```

The variables a, i0, i1, ... in are not visible to or accessible to x or the *embedded-statement* or any other source code of the program. The variable v is read-only in the embedded statement. If there is not an explicit conversion (23.5) from T (the element type) to V, an error is produced and no further steps are taken. If x has the value null, a System.NullReferenceException is thrown at run-time.

[*Note*: Although pointer types are not permitted as type arguments, pointer arrays may be used as type arguments. *end note*]

# 23.6 Pointers in expressions

## 23.6.1 General

In an unsafe context, an expression may yield a result of a pointer type, but outside an unsafe context, it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a compile-time error occurs if any *simple-name* (§12.7.3), *member-access* (§12.7.5), *invocation-expression* (§12.7.6), or *element-access* (§12.7.7) is of a pointer type.

In an unsafe context, the *primary-no-array-creation-expression* (§12.7) and *unary-expression* (§12.8) productions permit the following additional constructs:

primary-no-array-creation-expression:

*pointer-member-access pointer-element-access* 

unary-expression:

... pointer-indirection-expression addressof-expression

These constructs are described in the following subclauses.

[Note: The precedence and associativity of the unsafe operators is implied by the grammar. end note]

## 23.6.2 Pointer indirection

A pointer-indirection-expression consists of an asterisk (\*)followed by a unary-expression.

pointer-indirection-expression: \* unary-expression

The unary \* operator denotes pointer indirection and is used to obtain the variable to which a pointer points. The result of evaluating \*P, where P is an expression of a pointer type T\*, is a variable of type T. It is a compile-time error to apply the unary \* operator to an expression of type void\* or to an expression that isn't of a pointer type.

The effect of applying the unary \* operator to a null pointer is implementation-defined. In particular, there is no guarantee that this operation throws a System.NullReferenceException.

If an invalid value has been assigned to the pointer, the behavior of the unary \* operator is undefined. [*Note*: Among the invalid values for dereferencing a pointer by the unary \* operator are an address inappropriately aligned for the type pointed to (see example in §23.5), and the address of a variable after the end of its lifetime. *end note*]

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form \*P is considered initially assigned (§10.4.2).

# 23.6.3 Pointer member access

A *pointer-member-access* consists of a *primary-expression*, followed by a "->" token, followed by an *identifier* and an optional *type-argument-list*.

pointer-member-access: primary-expression -> identifier type-argument-list<sub>opt</sub>

In a pointer member access of the form P->I, P shall be an expression of a pointer type, and I shall denote an accessible member of the type to which P points.

A pointer member access of the form P->I is evaluated exactly as (\*P).I. For a description of the pointer indirection operator (\*), see §23.6.2. For a description of the member access operator (.), see §12.7.5.

[Example: In the following code

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

```
public override string ToString() {
    return "(" + x + "," + y + ")";
}
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.writeLine(p->ToString());
        }
    }
}
```

the -> operator is used to access fields and invoke a method of a struct through a pointer. Because the operation P->I is precisely equivalent to (\*P).I, the Main method could equally well have been written:

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

end example]

## 23.6.4 Pointer element access

A *pointer-element-access* consists of a *primary-no-array-creation-expression* followed by an expression enclosed in "[" and "]".

pointer-element-access: primary-no-array-creation-expression [ expression ]

In a pointer element access of the form P[E], P shall be an expression of a pointer type other than void\*, and E shall be an expression that can be implicitly converted to int, uint, long, or ulong.

A pointer element access of the form P[E] is evaluated exactly as \*(P + E). For a description of the pointer indirection operator (\*), see §23.6.2. For a description of the pointer addition operator (+), see §23.6.7.

[Example: In the following code

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}</pre>
```

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:

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}</pre>
```

end example]

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*]

## 23.6.5 The address-of operator

An *addressof-expression* consists of an ampersand (&) followed by a *unary-expression*.

addressof-expression: & unary-expression

Given an expression E which is of a type T and is classified as a fixed variable (§23.4), the construct &E computes the address of the variable given by E. The type of the result is T\* and is classified as a value. A compile-time error occurs if E is not classified as a variable, if E is classified as a read-only local variable, or if E denotes a moveable variable. In the last case, a fixed statement (§23.7) can be used to temporarily "fix" the variable before obtaining its address. [*Note*: As stated in §12.7.5, outside an instance constructor or static constructor for a struct or class that defines a readonly field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken. *end note*]

The & operator does not require its argument to be definitely assigned, but following an & operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

[Example: In the following code

```
using System;
class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.writeLine(i);
    }
}
```

i is considered definitely assigned following the &i operation used to initialize p. The assignment to \*p in effect initializes i, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed. *end example*]

[*Note*: The rules of definite assignment for the & operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure which is filled in by the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule, redundant initialization of the struct variable would be required. *end note*]

[*Note*: When a local variable, value parameter, or parameter array is captured by an anonymous function (§12.7.16), that local variable, parameter, or parameter array is no longer considered to be a fixed variable (§23.7), but is instead considered to be a moveable variable. Thus it is an error for any unsafe code to

take the address of a local variable, value parameter, or parameter array that has been captured by an anonymous function. *end note*]

#### 23.6.6 Pointer increment and decrement

In an unsafe context, the ++ and -- operators (§12.7.10 and §12.8.6) can be applied to pointer variables of all types except void\*. Thus, for every pointer type T\*, the following operators are implicitly defined:

```
T* operator ++(T* x);
```

T\* operator --(T\* x);

The operators produce the same results as x+1 and x-1, respectively (§23.6.7). In other words, for a pointer variable of type T\*, the ++ operator adds sizeof(T) to the address contained in the variable, and the -- operator subtracts sizeof(T) from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

#### 23.6.7 Pointer arithmetic

In an unsafe context, the + operator (§12.9.5) and - operator (§12.9.6) can be applied to values of all pointer types except void\*. Thus, for every pointer type T\*, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);
T* operator +(uint x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, ulong y);
T* operator -(T* x, ulong y);
long operator -(T* x, T* y);
```

Given an expression P of a pointer type T\* and an expression N of type int, uint, long, or ulong, the expressions P + N and N + P compute the pointer value of type T\* that results from adding N \* sizeof(T) to the address given by P. Likewise, the expression P - N computes the pointer value of type T\* that results from subtracting N \* sizeof(T) from the address given by P.

Given two expressions, P and Q, of a pointer type  $T^*$ , the expression P - Q computes the difference between the addresses given by P and Q and then divides that difference by sizeof(T). The type of the result is always long. In effect, P - Q is computed as ((long)(P) - (long)(Q)) / sizeof(T). [*Example*:

```
using System;
class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.writeLine("p - q = {0}", p - q);
            Console.writeLine("q - p = {0}", q - p);
        }
    }
}
```

which produces the output:

p - q = -14q - p = 14

end example]

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an implementation-defined fashion, but no exceptions are produced.

## 23.6.8 Pointer comparison

In an unsafe context, the ==, !=, <, >, <=, and >= operators (§12.11) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the void\* type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

## 23.6.9 The sizeof operator

For certain predefined types (§12.7.13), the sizeof operator yields a constant int value. For all other types, the result of the sizeof operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

# 23.7 The fixed statement

In an unsafe context, the *embedded-statement* (§13.1) production permits an additional construct, the fixed statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

```
embedded-statement:
...
fixed-statement
fixed-statement:
fixed ( pointer-type fixed-pointer-declarators ) embedded-statement
fixed-pointer-declarators:
    fixed-pointer-declarator , fixed-pointer-declarator
fixed-pointer-declarator:
    identifier = fixed-pointer-initializer
fixed-pointer-initializer:
    & variable-reference
    expression
```

Each *fixed-pointer-declarator* declares a local variable of the given *pointer-type* and initializes that local variable with the address computed by the corresponding *fixed-pointer-initializer*. A local variable declared in a fixed statement is accessible in any *fixed-pointer-initializers* occurring to the right of that variable's

declaration, and in the *embedded-statement* of the fixed statement. A local variable declared by a fixed statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the ++ and -- operators) or pass it as a ref or out parameter.

It is an error to use a captured local variable (§12.16.6.2), value parameter, or parameter array in a *fixed*-*pointer-initializer* can be one of the following:

- The token "&" followed by a *variable-reference* (§10.5) to a moveable variable (§23.4) of an unmanaged type T, provided the type T\* is implicitly convertible to the pointer type given in the fixed statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the fixed statement.
- An expression of an *array-type* with elements of an unmanaged type T, provided the type T\* is implicitly convertible to the pointer type given in the fixed statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the fixed statement. The behavior of the fixed statement is implementation-defined if the array expression is null or if the array has zero elements.
- An expression of type string, provided the type char\* is implicitly convertible to the pointer type given in the fixed statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the fixed statement. The behavior of the fixed statement is implementation-defined if the string expression is null.
- A simple-name or member-access that references a fixed-size buffer member of a moveable variable, provided the type of the fixed-size buffer member is implicitly convertible to the pointer type given in the fixed statement. In this case, the initializer computes a pointer to the first element of the fixed-size buffer (§23.8.3), and the fixed-size buffer is guaranteed to remain at a fixed address for the duration of the fixed statement.

For each address computed by a *fixed-pointer-initializer* the fixed statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the fixed statement. [*Example*: If the address computed by a *fixed-pointer-initializer* references a field of an object or an element of an array instance, the fixed statement guarantees that the containing object instance is not relocated or disposed of during the lifetime of the statement. *end example*]

It is the programmer's responsibility to ensure that pointers created by fixed statements do not survive beyond execution of those statements. [*Example*: When pointers created by fixed statements are passed to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these pointers. *end example*]

Fixed objects can cause fragmentation of the heap (because they can't be moved). For that reason, objects should be fixed only when absolutely necessary and then only for the shortest amount of time possible. [*Example*: The example

```
class Test
{
  static int x;
  int y;
  unsafe static void F(int* p) {
     *p = 1;
  }
}
```

```
static void Main() {
    Test t = new Test();
    int[] a = new int[10];
    unsafe {
        fixed (int* p = &x) F(p);
        fixed (int* p = &t.y) F(p);
        fixed (int* p = &a[0]) F(p);
        fixed (int* p = a) F(p);
     }
}
```

demonstrates several uses of the fixed statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case, it would have been an error to use the regular & operator since the variables are all classified as moveable variables.

The third and fourth fixed statements in the example above produce identical results. In general, for an array instance a, specifying &a[0] in a fixed statement is the same as simply specifying a.Here's another example of the fixed statement, this time using string:

```
class Test
{
  static string name = "xx";
  unsafe static void F(char* p) {
    for (int i = 0; p[i] != '\0'; ++i)
        Console.WriteLine(p[i]);
  }
  static void Main() {
    unsafe {
        fixed (char* p = name) F(p);
        fixed (char* p = "xx") F(p);
        }
   }
}
```

## end example]

In an unsafe context, array elements of single-dimensional arrays are stored in increasing index order, starting with index 0 and ending with index Length -1. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

Within a fixed statement that obtains a pointer p to an array instance a, the pointer values ranging from p to p + a.Length – 1 represent addresses of the elements in the array. Likewise, the variables ranging from p[0] to p[a.Length – 1] represent the actual array elements. Given the way in which arrays are stored, we can treat an array of any dimension as though it were linear. [*Example*:

which produces the output:

}

[0,0,0] = 0	[0,0,1] = 1	[0,0,2] = 2	[0,0,3] = 3
[0,1,0] = 4	[0,1,1] = 5	[0,1,2] = 6	[0,1,3] = 7
[0,2,0] = 8	[0,2,1] = 9	[0,2,2] = 10	[0,2,3] = 11
[1,0,0] = 12	[1,0,1] = 13	[1,0,2] = 14	[1,0,3] = 15
[1,1,0] = 16	[1,1,1] = 17	[1,1,2] = 18	[1,1,3] = 19
[1,2,0] = 20	[1,2,1] = 21	[1,2,2] = 22	[1,2,3] = 23

end example]

[Example: In the following code

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
      for (; count != 0; count--) *p++ = value;
    }
    static void Main() {
      int[] a = new int[100];
      unsafe {
        fixed (int* p = a) Fill(p, 100, -1);
      }
    }
}
```

a fixed statement is used to fix an array so its address can be passed to a method that takes a pointer. end example]

A char\* value produced by fixing a string instance always points to a null-terminated string. Within a fixed statement that obtains a pointer p to a string instance s, the pointer values ranging from p to

p + s.Length - 1 represent addresses of the characters in the string, and the pointer value

p + s.Length always points to a null character (the character with value '\0').

Modifying objects of managed type through fixed pointers can result in undefined behavior. [*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*]

[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*]

## 23.8 Fixed-size buffers

#### 23.8.1 General

Fixed-size buffers are used to declare "C-style" in-line arrays as members of structs, and are primarily useful for interfacing with unmanaged APIs.

## 23.8.2 Fixed-size buffer declarations

A *fixed-size buffer* is a member that represents storage for a fixed-length buffer of variables of a given type. A fixed-size buffer declaration introduces one or more fixed-size buffers of a given element type. Fixed-size buffers are only permitted in struct declarations and may only occur in unsafe contexts (§23.2).

struct-member-declaration: fixed-size-buffer-declaration fixed-size-buffer-declaration: attributes<sub>opt</sub> fixed-size-buffer-modifiers<sub>opt</sub> fixed buffer-element-type fixed-size-buffer-declarators ; fixed-size-buffer-modifiers: fixed-size-buffer-modifier fixed-size-buffer-modifier fixed-size-buffer-modifiers fixed-size-buffer-modifier: new public protected internal private unsafe buffer-element-type: type fixed-size-buffer-declarators: fixed-size-buffer-declarator fixed-size-buffer-declarator , fixed-size-buffer-declarators *fixed-size-buffer-declarator: identifier* [ *constant-expression* ]

A fixed-size buffer declaration may include a set of attributes (§22), a new modifier (§15.3.5), a valid combination of the four access modifiers (§15.3.6) and an unsafe modifier (§23.2). The attributes and modifiers apply to all of the members declared by the fixed-size buffer declaration. It is an error for the same modifier to appear multiple times in a fixed-size buffer declaration.

A fixed-size buffer declaration is not permitted to include the static modifier.

The buffer element type of a fixed-size buffer declaration specifies the element type of the buffer(s) introduced by the declaration. The buffer element type shall be one of the predefined types sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or bool.

The buffer element type is followed by a list of fixed-size buffer declarators, each of which introduces a new member. A fixed-size buffer declarator consists of an identifier that names the member, followed by a constant expression enclosed in [ and ] tokens. The constant expression denotes the number of elements in the member introduced by that fixed-size buffer declarator. The type of the constant expression shall be implicitly convertible to type int, and the value shall be a non-zero positive integer.

The elements of a fixed-size buffer shall be laid out sequentially in memory.

A fixed-size buffer declaration that declares multiple fixed-size buffers is equivalent to multiple declarations of a single fixed-size buffer declation with the same attributes, and element types. [*Example*:

```
unsafe struct A
{
    public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
    public fixed int x[5];
    public fixed int y[10];
    public fixed int z[100];
}
```

end example]

## 23.8.3 Fixed-size buffers in expressions

Member lookup (§12.5) of a fixed-size buffer member proceeds exactly like member lookup of a field.

A fixed-size buffer can be referenced in an expression using a *simple-name* (§12.6.3) or a *member-access* (§12.6.5).

When a fixed-size buffer member is referenced as a simple name, the effect is the same as a member access of the form this.I, where I is the fixed-size buffer member.

In a member access of the form E.I, if E is of a struct type and a member lookup of I in that struct type identifies a fixed-size member, then E.I is evaluated an classified as follows:

- If the expression E.I does not occur in an unsafe context, a compile-time error occurs.
- If E is classified as a value, a compile-time error occurs.
- Otherwise, if E is a moveable variable (§23.4) and the expression E.I is not a *fixed-pointer-initializer* (§23.7), a compile-time error occurs.
- Otherwise, E references a fixed variable and the result of the expression is a pointer to the first element of the fixed-size buffer member I in E. The result is of type S\*, where S is the element type of I, and is classified as a value.

The subsequent elements of the fixed-size buffer can be accessed using pointer operations from the first element. Unlike access to arrays, access to the elements of a fixed-size buffer is an unsafe operation and is not range checked.

[Example: The following declares and uses a struct with a fixed-size buffer member.

```
unsafe struct Font
ſ
   public int size;
   public fixed char name[32];
}
class Test
   unsafe static void PutString(string s, char* buffer, int bufSize) {
       int len = s.Length;
       if (len > bufSize) len = bufSize;
       for (int i = 0; i < len; i++) buffer[i] = s[i];
for (int i = len; i < bufSize; i++) buffer[i] = (char)0;</pre>
   }
   unsafe static void Main()
       Font f;
       f.size = 10;
       PutString("Times New Roman", f.name, 32);
   }
}
```

end example]

## 23.8.4 Definite assignment checking

Fixed-size buffers are not subject to definite assignment-checking (§10.4), and fixed-size buffer members are ignored for purposes of definite-assignment checking of struct type variables.

When the outermost containing struct variable of a fixed-size buffer member is a static variable, an instance variable of a class instance, or an array element, the elements of the fixed-size buffer are automatically initialized to their default values (§10.3). In all other cases, the initial content of a fixed-size buffer is undefined.

# 23.9 Stack allocation

In an unsafe context, a local variable declaration (§13.6.2) may include a stack allocation initializer, which allocates memory from the call stack.

local-variable-initializer: ... stackalloc-initializer stackalloc-initializer: stackalloc unmanaged-type [ expression ]

The *unmanaged-type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a *constant-expression* that evaluates to a negative value.

A stack allocation initializer of the form stackalloc T[E] requires T to be an unmanaged type (§23.3) and E to be an expression implicitly convertible to type int. The construct allocates E \* sizeof(T) bytes from the call stack and returns a pointer, of type T\*, to the newly allocated block. If E is a negative value, then the behavior is undefined. If E is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a System.StackOverflowException is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in catch or finally blocks (§13.11).

[*Note*: There is no way to explicitly free memory allocated using stackalloc. *end note*] All stack-allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. [*Note*: This corresponds to the alloca function, an extension commonly found in C and C++ implementations. *end note*]

[Example: In the following code

```
using System;
class Test
   static string IntToString(int value) {
       int n = value >= 0 ? value : -value;
      unsafe {
    char* buffer = stackalloc char[16];
          char* p = buffer + 16;
          do {
               --p = (char)(n % 10 + '0');
           n /= 10;
while (n != 0);
          } while (n != v);
if (value < 0) *--p = '-'</pre>
          return new string(p, 0, (int)(buffer + 16 - p));
      }
   }
   static void Main() {
      Console.WriteLine(IntToString(12345));
      Console.WriteLine(IntToString(-999));
   }
}
```

a stackalloc initializer is used in the IntToString method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns. *end example*]

Except for the stackalloc operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system.

End of conditionally normative text.

# Annex A. Grammar

This clause is informative.

## A.1 General

This annex contains summaries of the lexical and syntactic grammars found in the main document, and of the grammar extensions for unsafe code. Grammar productions appear here in the same order that they appear in the main document.

#### A.2 Lexical grammar

## input:: input-section<sub>opt</sub> input-section:: input-section-part input-section input-section-part input-section-part:: input-elements<sub>opt</sub> new-line pp-directive input-elements:: input-element input-elements input-element input-element:: whitespace comment token Line terminators new-line:: Carriage return character (U+000D) Line feed character (U+000A) Carriage return character (U+000D) followed by line feed character (U+000A) *Next line character* (U+0085) *Line separator character* (U+2028) Paragraph separator character (U+2029) White space whitespace:: whitespace-character whitespace whitespace-character whitespace-character:: Any character with Unicode class Zs Horizontal tab character (U+0009) Vertical tab character (U+000B) *Form feed character* (U+000C)

## A.2.1 Comments

comment:: single-line-comment delimited-comment

single-line-comment:: // input-characters<sub>opt</sub>

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

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

new-line-character::

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

#### delimited-comment::

/\* delimited-comment-text<sub>opt</sub> asterisks /

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

delimited-comment-section::

```
/
asterisks<sub>opt</sub> not-slash-or-asterisk
```

asterisks::

\*

asterisks \*

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

# A.2.2 Tokens

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

Unicode character escape sequences

unicode-escape-sequence::

- \u hex-digit hex-digit hex-digit hex-digit
- \U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

Identifiers

identifier:: available-identifier @ identifier-or-keyword

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

identifier-or-keyword::

identifier-start-character identifier-part-charactersopt

identifier-start-character:: letter-character underscore-character

underscore-character::

\_ (the underscore character U+005F) A unicode-escape-sequence representing the character U+005F

identifier-part-characters::

identifier-part-character identifier-part-characters identifier-part-character

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

letter-character::

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

combining-character::

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

decimal-digit-character::

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

connecting-character::

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

formatting-character::

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

## A.2.3 Keywords

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

#### LILEIUIS

literal:: boolean-literal integer-literal real-literal character-literal string-literal null-literal boolean-literal:: true false integer-literal:: decimal-integer-literal hexadecimal-integer-literal decimal-integer-literal:: decimal-digits integer-type-suffixopt decimal-digits:: decimal-digit decimal-digits decimal-digit decimal-digit:: one of 0 1 2 3 4 5 6 7 8 9 integer-type-suffix:: one of U u L I UL UI uL uI LU Lu IU Iu hexadecimal-integer-literal:: 0x hex-digits integer-type-suffix<sub>opt</sub> 0X hex-digits integer-type-suffix<sub>opt</sub> hex-digits:: hex-digit hex-digits hex-digit hex-digit:: one of 0123456789ABCDEFabcdef

```
real-literal::
    decimal-digits . decimal-digits exponent-part<sub>opt</sub> real-type-suffix<sub>opt</sub>
    . decimal-digits exponent-part<sub>opt</sub> real-type-suffix<sub>opt</sub>
    decimal-digits exponent-part real-type-suffixopt
    decimal-digits real-type-suffix
exponent-part::
    e signopt decimal-digits
    E signopt decimal-digits
sign:: one of
    + -
real-type-suffix:: one of
    F f D d M m
character-literal::
    ' character '
character::
    single-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence
single-character::
    Any character except ' (U+0027), \ (U+005C), and new-line-character
simple-escape-sequence:: one of
    ' '' \setminus 0 \langle a \rangle f \langle n \rangle r \langle t \rangle
hexadecimal-escape-sequence:
    x hex-digit hex-digit<sub>opt</sub> hex-digit<sub>opt</sub> hex-digit<sub>opt</sub>
string-literal::
    regular-string-literal
    verbatim-string-literal
regular-string-literal::
    " regular-string-literal-characters<sub>opt</sub> "
regular-string-literal-characters::
    regular-string-literal-character
    regular-string-literal-characters regular-string-literal-character
regular-string-literal-character::
    single-regular-string-literal-character
    simple-escape-sequence
    hexadecimal-escape-sequence
    unicode-escape-sequence
single-regular-string-literal-character::
    Any character except " (U+0022), \setminus (U+005C), and new-line-character
verbatim-string-literal::
    @" verbatim-string-literal-characters<sub>opt</sub> "
verbatim-string-literal-characters::
    verbatim-string-literal-character
    verbatim-string-literal-characters verbatim-string-literal-character
verbatim-string-literal-character::
    single-verbatim-string-literal-character
    quote-escape-sequence
```

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

quote-escape-sequence::

null-literal:: null

## A.2.4 Operators and punctuators

operator-or-punctuator:: one of

{	}	Γ	]	(	)		,	:	;
+	-	*	/	%	&		٨	!	~
=	<	>	?	??	::	++		&&	
->	==	!=	<=	>=	+=	-=	*=	/=	%=
&=	=	$\wedge =$	<<	<<=					

right-shift::

> >

right-shift-assignment::

> >=

#### A.2.5 Pre-processing directives

```
pp-directive::
    pp-declaration
    pp-conditional
    pp-line
    pp-diagnostic
    pp-region
    pp-pragma
conditional-symbol::
    Any identifier-or-keyword except true or false
pp-expression::
    whitespace<sub>opt</sub> pp-or-expression whitespace<sub>opt</sub>
pp-or-expression::
    pp-and-expression
    pp-or-expression whitespace<sub>opt</sub> || whitespace<sub>opt</sub> pp-and-expression
pp-and-expression::
    pp-equality-expression
    pp-and-expression whitespace<sub>opt</sub> && whitespace<sub>opt</sub> pp-equality-expression
pp-equality-expression::
    pp-unary-expression
    pp-equality-expression whitespace<sub>opt</sub> == whitespace<sub>opt</sub> pp-unary-expression
    pp-equality-expression whitespace<sub>opt</sub> != whitespace<sub>opt</sub> pp-unary-expression
pp-unary-expression::
    pp-primary-expression
    ! whitespaceopt pp-unary-expression
pp-primary-expression::
    true
    false
    conditional-symbol
    ( whitespace<sub>opt</sub> pp-expression whitespace<sub>opt</sub> )
```

pp-declaration:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> define whitespace conditional-symbol pp-new-line whitespace<sub>opt</sub> # whitespace<sub>opt</sub> undef whitespace conditional-symbol pp-new-line pp-new-line:: whitespace<sub>opt</sub> single-line-comment<sub>opt</sub> new-line pp-conditional:: pp-if-section pp-elif-sections<sub>opt</sub> pp-else-section<sub>opt</sub> pp-endif pp-if-section:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> if whitespace pp-expression pp-new-line conditional-section<sub>opt</sub> pp-elif-sections:: pp-elif-section pp-elif-sections pp-elif-section pp-elif-section:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> elif whitespace pp-expression pp-new-line conditional-section<sub>opt</sub> pp-else-section:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> else pp-new-line conditional-section<sub>opt</sub> pp-endif:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> endif pp-new-line conditional-section:: input-section skipped-section skipped-section:: skipped-section-part skipped-section skipped-section-part skipped-section-part:: skipped-characters<sub>opt</sub> new-line pp-directive skipped-characters:: whitespace<sub>opt</sub> not-number-sign input-characters<sub>opt</sub> not-number-sign:: Any input-character except # pp-line:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> line whitespace line-indicator pp-new-line line-indicator:: decimal-digits whitespace file-name decimal-digits default hidden file-name:: " file-name-characters " file-name-characters:: file-name-character file-name-characters file-name-character file-name-character:: Any input-character except " (U+0022), and new-line-character

#### ECMA-334

pp-diagnostic:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> error pp-message whitespace<sub>opt</sub> # whitespace<sub>opt</sub> warning pp-message pp-message:: new-line whitespace input-characters<sub>opt</sub> new-line pp-region:: pp-start-region conditional-section<sub>opt</sub> pp-end-region pp-start-region:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> region pp-message pp-end-region:: whitespace<sub>opt</sub> # whitespace<sub>opt</sub> endregion pp-message pp-pragma:: whitespaceopt # whitespaceopt pragma pp-pragma-text pp-pragma-text:: new-line whitespace input-charactersopt new-line

## A.3 Syntactic grammar

#### A.3.1 Basic concepts

namespace-name: namespace-or-type-name

type-name: namespace-or-type-name

namespace-or-type-name:

identifier type-argument-list<sub>opt</sub> namespace-or-type-name . identifier type-argument-list<sub>opt</sub> qualified-alias-member

## A.3.2 Types

type: reference-type value-type type-parameter

value-type: struct-type enum-type

struct-type: type-name simple-type nullable-value-type

simple-type: numeric-type bool

numeric-type: integral-type floating-point-type decimal integral-type: sbyte byte short ushort int uint long ulong char nullable-type: non-nullable-value-type ? non-nullable-value-type: type floating-point-type: float double enum-type: type-name type-argument-list: < type-arguments > type-arguments: type-argument type-arguments, type-argument type-argument: type type-parameter: identifier A.3.3 Variables variable-reference: expression A.3.4 Expressions argument-list: argument argument-list , argument argument: argument-name<sub>opt</sub> argument-value argument-name: identifier : argument-value: expression ref variable-reference out variable-reference

primary-expression: primary-no-array-creation-expression array-creation-expression primary-no-array-creation-expression: literal simple-name parenthesized-expression member-access invocation-expression element-access this-access base-access post-increment-expression post-decrement-expression object-creation-expression delegate-creation-expression anonymous-object-creation-expression typeof-expression sizeof-expression checked-expression unchecked-expression default-value-expression anonymous-method-expression simple-name: identifier type-argument-listopt parenthesized-expression: ( expression ) member-access: primary-expression . identifier type-argument-listopt predefined-type . identifier type-argument-listopt qualified-alias-member . identifier type-argument-listopt predefined-type: one of decimal double float boo1 byte char int long object sbyte short string uint ulong ushort invocation-expression: primary-expression ( argument-list<sub>opt</sub> ) element-access: primary-no-array-creation-expression [ argument-list ] expression-list: expression expression-list , expression this-access: this base-access: base . identifier type-argument-listopt base [ argument-list ] *post-increment-expression:* primary-expression ++ post-decrement-expression: primary-expression --

```
object-creation-expression:
    new type ( argument-list<sub>opt</sub> ) object-or-collection-initializer<sub>opt</sub>
    new type object-or-collection-initializer
object-or-collection-initializer:
    object-initializer
    collection-initializer
object-initializer:
    { member-initializer-list<sub>opt</sub> }
    { member-initializer-list , }
member-initializer-list:
    member-initializer
    member-initializer-list , member-initializer
member-initializer:
    identifier = initializer-value
initializer-value:
    expression
    object-or-collection-initializer
collection-initializer:
    { element-initializer-list }
    { element-initializer-list , }
element-initializer-list:
    element-initializer
    element-initializer-list , element-initializer
element-initializer:
    non-assignment-expression
    { expression-list }
array-creation-expression:
    new non-array-type [ expression-list ] rank-specifiers<sub>opt</sub> array-initializer<sub>opt</sub>
    new array-type array-initializer
    new rank-specifier array-initializer
delegate-creation-expression:
    new delegate-type ( expression )
anonymous-object-creation-expression:
    new anonymous-object-initializer
anonymous-object-initializer:
    { member-declarator-list<sub>opt</sub> }
    { member-declarator-list , }
member-declarator-list:
    member-declarator
    member-declarator-list , member-declarator
member-declarator:
    simple-name
    member-access
    base-access
    identifier = expression
```

```
typeof-expression:
   typeof ( type )
   typeof ( unbound-type-name )
   typeof ( void )
unbound-type-name:
   identifier generic-dimension-specifier<sub>opt</sub>
   identifier :: identifier generic-dimension-specifier<sub>opt</sub>
   unbound-type-name . identifier generic-dimension-specifier<sub>opt</sub>
generic-dimension-specifier:
   < commas<sub>opt</sub> >
commas:
   commas ,
checked-expression:
   checked (expression)
unchecked-expression:
   unchecked ( expression )
default-value-expression:
   default ( type )
unary-expression:
   primary-expression
   + unary-expression
   - unary-expression
    ! unary-expression
   ~ unary-expression
   pre-increment-expression
   pre-decrement-expression
   cast-expression
   await-expression
pre-increment-expression:
   ++ unary-expression
pre-decrement-expression:
    -- unary-expression
cast-expression:
    (type) unary-expression
await-expression:
   await unary-expression
multiplicative-expression:
   unary-expression
   multiplicative-expression * unary-expression
   multiplicative-expression / unary-expression
   multiplicative-expression % unary-expression
additive-expression:
   multiplicative-expression
   additive-expression + multiplicative-expression
   additive-expression - multiplicative-expression
```

shift-expression: additive-expression shift-expression << additive-expression shift-expression right-shift additive-expression
relational-expression: shift-expression relational-expression < shift-expression relational-expression > shift-expression relational-expression <= shift-expression relational-expression >= shift-expression relational-expression is type relational-expression as type
equality-expression: relational-expression equality-expression == relational-expression equality-expression != relational-expression
and-expression: equality-expression and-expression & equality-expression
exclusive-or-expression: and-expression exclusive-or-expression ^ and-expression
inclusive-or-expression: exclusive-or-expression inclusive-or-expression   exclusive-or-expression
conditional-and-expression: inclusive-or-expression conditional-and-expression && inclusive-or-expression
conditional-or-expression: conditional-and-expression conditional-or-expression    conditional-and-expression
null-coalescing-expression: conditional-or-expression conditional-or-expression ?? null-coalescing-expression
conditional-expression: null-coalescing-expression null-coalescing-expression ? expression : expression
lambda-expression: async <sub>opt</sub> anonymous-function-signature => anonymous-function-body
anonymous-method-expression: async <sub>opt</sub> delegate explicit-anonymous-function-signature <sub>opt</sub> block
anonymous-function-signature: explicit-anonymous-function-signature implicit-anonymous-function-signature
explicit-anonymous-function-signature: ( explicit-anonymous-function-parameter-listopt )
explicit-anonymous-function-parameter-list: explicit-anonymous-function-parameter explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter

explicit-anonymous-function-parameter: anonymous-function-parameter-modifier<sub>opt</sub> type identifier anonymous-function-parameter-modifier: ref out implicit-anonymous-function-signature: ( *implicit-anonymous-function-parameter-list*<sub>opt</sub> ) implicit-anonymous-function-parameter implicit-anonymous-function-parameter-list: implicit-anonymous-function-parameter implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter implicit-anonymous-function-parameter: identifier anonymous-function-body: expression block query-expression: from-clause query-body from-clause: from type<sub>opt</sub> identifier in expression query-body: query-body-clauses<sub>opt</sub> select-or-group-clause query-continuation<sub>opt</sub> query-body-clauses: query-body-clause query-body-clauses query-body-clause query-body-clause: from-clause let-clause where-clause join-clause join-into-clause orderby-clause let-clause: let identifier = expression where-clause: where boolean-expression join-clause: join type<sub>opt</sub> identifier in expression on expression equals expression join-into-clause: join typeopt identifier in expression on expression equals expression into identifier orderby-clause: orderby orderings orderings: ordering orderings, ordering

ordering: expression ordering-direction<sub>opt</sub> ordering-direction: ascending descending select-or-group-clause: select-clause group-clause select-clause: select *expression* group-clause: group expression by expression query-continuation: into identifier query-body assignment: unary-expression assignment-operator expression assignment-operator: = += -= \*= /= %= &= |= ^= <<= right-shift-assignment expression: non-assignment-expression assignment non-assignment-expression: conditional-expression lambda-expression query-expression constant-expression: expression boolean-expression: expression A.3.5 Statements statement: labeled-statement

labeled-statement declaration-statement embedded-statement embedded-statement: block empty-statement *expression-statement* selection-statement iteration-statement jump-statement try-statement checked-statement unchecked-statement lock-statement using-statement yield-statement block: { statement-list<sub>opt</sub> } statement-list: statement statement-list statement empty-statement: ; labeled-statement: identifier : statement declaration-statement: *local-variable-declaration* ; local-constant-declaration ; local-variable-declaration: local-variable-type local-variable-declarators local-variable-type: type var local-variable-declarators: local-variable-declarator local-variable-declarators , local-variable-declarator local-variable-declarator: identifier identifier = local-variable-initializer local-variable-initializer: expression array-initializer local-constant-declaration: const type constant-declarators constant-declarators: constant-declarator constant-declarators , constant-declarator constant-declarator: identifier = constant-expression *expression-statement:* statement-expression ;

statement-expression: invocation-expression object-creation-expression assignment post-increment-expression post-decrement-expression pre-increment-expression pre-decrement-expression await-expression selection-statement: *if-statement* switch-statement *if-statement:* if ( boolean-expression ) embedded-statement if ( boolean-expression ) embedded-statement else embedded-statement switch-statement: switch ( expression ) switch-block switch-block: { switch-sections<sub>opt</sub> } switch-sections: switch-section switch-sections switch-section switch-section: switch-labels statement-list switch-labels: switch-label switch-labels switch-label switch-label: case constant-expression : default : iteration-statement: while-statement do-statement for-statement foreach-statement while-statement: while ( boolean-expression ) embedded-statement do-statement: do embedded-statement while ( boolean-expression ) ; for-statement: for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement for-initializer: local-variable-declaration statement-expression-list for-condition: boolean-expression for-iterator: statement-expression-list

statement-expression-list: statement-expression statement-expression-list , statement-expression foreach-statement: foreach ( local-variable-type identifier in expression ) embedded-statement jump-statement: break-statement continue-statement goto-statement return-statement throw-statement break-statement: break ; continue-statement: continue ; goto-statement: goto identifier ; goto case constant-expression ; goto default ; return-statement: return *expression*<sub>opt</sub>; throw-statement: throw *expression*<sub>opt</sub> ; try-statement: try block catch-clauses try block catch-clauses<sub>opt</sub> finally-clause catch-clauses: specific-catch-clauses specific-catch-clauses<sub>opt</sub> general-catch-clause specific-catch-clauses: specific-catch-clause specific-catch-clauses specific-catch-clause *specific-catch-clause:* catch ( type identifier<sub>opt</sub> ) block *general-catch-clause:* catch block finally-clause: finally *block* checked-statement: checked block unchecked-statement: unchecked block lock-statement: lock ( expression ) embedded-statement using-statement: using ( resource-acquisition ) embedded-statement

resource-acquisition: local-variable-declaration expression yield-statement: yield return expression ; yield break ; A.3.6 Namespaces compilation-unit: extern-alias-directives<sub>opt</sub> using-directives<sub>opt</sub> global-attributes<sub>opt</sub> namespace-member-declarations<sub>opt</sub> namespace-declaration: namespace qualified-identifier namespace-body ; opt qualified-identifier: identifier qualified-identifier . identifier namespace-body: { extern-alias-directives<sub>opt</sub> using-directives<sub>opt</sub> namespace-member-declarations<sub>opt</sub> } extern-alias-directives: extern-alias-directive extern-alias-directives extern-alias-directive extern-alias-directive: extern alias identifier ; using-directives: using-directive using-directives using-directive using-directive: using-alias-directive using-namespace-directive using-alias-directive: using identifier = namespace-or-type-name ; using-namespace-directive: using *namespace-name* ; namespace-member-declarations: namespace-member-declaration namespace-member-declarations namespace-member-declaration namespace-member-declaration: namespace-declaration type-declaration type-declaration: class-declaration struct-declaration interface-declaration enum-declaration delegate-declaration qualified-alias-member: identifier :: identifier type-argument-listopt

#### A.3.7 Classes

```
class-declaration:
    attributes<sub>opt</sub> class-modifiers<sub>opt</sub> partial<sub>opt</sub> class identifier type-parameter-list<sub>opt</sub>
            class-base<sub>opt</sub> type-parameter-constraints-clauses<sub>opt</sub> class-body ;<sub>opt</sub>
class-modifiers:
    class-modifier
    class-modifiers class-modifier
class-modifier:
    new
    public
   protected
    internal
    private
    abstract
    sealed
    static
type-parameter-list:
    < type-parameters >
type-parameters:
    attributes<sub>opt</sub> type-parameter
    type-parameters , attributes<sub>opt</sub> type-parameter
class-base:
    : class-type
    : interface-type-list
    : class-type , interface-type-list
interface-type-list:
    interface-type
    interface-type-list , interface-type
type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses type-parameter-constraints-clause
type-parameter-constraints-clause:
    where type-parameter : type-parameter-constraints
type-parameter-constraints:
   primary-constraint
   secondary-constraints
   constructor-constraint
   primary-constraint, secondary-constraints
    primary-constraint , constructor-constraint
   secondary-constraints, constructor-constraint
   primary-constraint, secondary-constraints, constructor-constraint
primary-constraint:
   class-type
    class
    struct
secondary-constraints:
    interface-type
    type-parameter
    secondary-constraints , interface-type
```

secondary-constraints, type-parameter

```
constructor-constraint:
    new ()
class-body:
    { class-member-declarations<sub>opt</sub> }
class-member-declarations:
    class-member-declaration
    class-member-declarations class-member-declaration
class-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    finalizer-declaration
    static-constructor-declaration
    type-declaration
constant-declaration:
    attributes<sub>opt</sub> constant-modifiers<sub>opt</sub> const type constant-declarators ;
constant-modifiers:
    constant-modifier
    constant-modifiers constant-modifier
constant-modifier:
    new
    public
    protected
    internal
    private
constant-declarators:
    constant-declarator
    constant-declarators, constant-declarator
constant-declarator:
    identifier = constant-expression
field-declaration:
    attributes<sub>opt</sub> field-modifiers<sub>opt</sub> type variable-declarators ;
field-modifiers:
   field-modifier
   field-modifiers field-modifier
field-modifier:
    new
    public
    protected
    internal
    private
    static
    readonly
    volatile
```

variable-declarators: variable-declarator variable-declarators, variable-declarator variable-declarator: identifier *identifier = variable-initializer* variable-initializer: expression array-initializermethod-declaration: *method-header method-body method-header:* attributes<sub>opt</sub> method-modifiers<sub>opt</sub> partial<sub>opt</sub> return-type member-name type-parameter-list<sub>opt</sub> ( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub> method-modifiers: method-modifier method-modifiers method-modifier method-modifier: new public protected internal private static virtual sealed override abstract extern async return-type: type void method-body: block ; formal-parameter-list: fixed-parameters fixed-parameters , parameter-array parameter-array fixed-parameters: fixed-parameter fixed-parameters , fixed-parameter fixed-parameter: attributes<sub>opt</sub> parameter-modifier<sub>opt</sub> type identifier default-argument<sub>opt</sub> default-argument: = expression

parameter-modifier: parameter-mode-modifier this parameter-mode-modifier: ref out parameter-array: attributes<sub>opt</sub> params array-type identifier property-declaration: attributes<sub>opt</sub> property-modifiers<sub>opt</sub> type member-name { accessor-declarations } property-modifiers: property-modifier property-modifiers property-modifier property-modifier: new public protected internal private static virtual sealed override abstract extern accessor-declarations: get-accessor-declaration set-accessor-declaration<sub>opt</sub> set-accessor-declaration get-accessor-declaration<sub>opt</sub> get-accessor-declaration: attributes<sub>opt</sub> accessor-modifier<sub>opt</sub> get accessor-body set-accessor-declaration: attributes<sub>opt</sub> accessor-modifier<sub>opt</sub> set accessor-body accessor-modifier: protected internal private protected internal internal protected accessor-body: block ; event-declaration: attributes<sub>opt</sub> event-modifiers<sub>opt</sub> event type variable-declarators ; attributes<sub>opt</sub> event-modifiers<sub>opt</sub> event type member-name { event-accessor-declarations } event-modifiers: event-modifier event-modifiers event-modifier

```
event-modifier:
   new
   public
   protected
   internal
   private
   static
   virtual
   sealed
   override
   abstract
   extern
event-accessor-declarations:
   add-accessor-declaration remove-accessor-declaration
   remove-accessor-declaration add-accessor-declaration
add-accessor-declaration:
   attributesopt add block
remove-accessor-declaration:
   attributes<sub>opt</sub> remove block
indexer-declaration:
   attributes<sub>opt</sub> indexer-modifiers<sub>opt</sub> indexer-declarator { accessor-declarations }
indexer-modifiers:
   indexer-modifier
   indexer-modifiers indexer-modifier
indexer-modifier:
   new
   public
   protected
   internal
   private
   virtual
   sealed
   override
   abstract
   extern
indexer-declarator:
   type this [ formal-parameter-list ]
   type interface-type . this [ formal-parameter-list ]
operator-declaration:
   attributes<sub>opt</sub> operator-modifiers operator-declarator operator-body
operator-modifiers:
   operator-modifier
   operator-modifiers operator-modifier
operator-modifier:
   public
   static
   extern
```

operator-declarator: unary-operator-declarator binary-operator-declarator conversion-operator-declarator unary-operator-declarator: type operator overloadable-unary-operator ( fixed-parameter ) overloadable-unary-operator: one of false - ! ~ ++ true + -*binary-operator-declarator:* type operator overloadable-binary-operator (fixed-parameter, fixed-parameter) overloadable-binary-operator: one of \_ \* & ٨ right-shift % << / + != == > < >= <= conversion-operator-declarator: implicit operator type ( fixed-parameter ) explicit operator type ( fixed-parameter ) operator-body: block ; constructor-declaration: attributes<sub>opt</sub> constructor-modifiers<sub>opt</sub> constructor-declarator constructor-body constructor-modifiers: constructor-modifier constructor-modifiers constructor-modifier constructor-modifier: public protected internal private extern constructor-declarator: identifier ( formal-parameter-list<sub>opt</sub> ) constructor-initializer<sub>opt</sub> constructor-initializer: : base ( *argument-list*<sub>opt</sub> ) : this ( argument-list<sub>opt</sub> ) constructor-body: block ; static-constructor-declaration: attributes<sub>opt</sub> static-constructor-modifiers identifier () static-constructor-body static-constructor-modifiers: externopt static static extern<sub>opt</sub> static-constructor-body: block ;

```
finalizer-declaration:
    attributes<sub>opt</sub> extern<sub>opt</sub> ~ identifier ( ) finalizer-body
finalizer-body:
    block
;
```

# A.3.8 Structs

struct-declaration:

```
attributes<sub>opt</sub> struct-modifiers<sub>opt</sub> partial<sub>opt</sub> struct identifier type-parameter-list<sub>opt</sub>
struct-interfaces<sub>opt</sub> type-parameter-constraints-clauses<sub>opt</sub> struct-body ;<sub>opt</sub>
```

struct-modifiers: struct-modifier struct-modifiers struct-modifier

struct-modifier:

new public protected internal private

struct-interfaces: : interface-type-list

struct-body:

{ struct-member-declarations<sub>opt</sub> }

struct-member-declarations: struct-member-declaration

struct-member-declaration struct-member-declaration

struct-member-declaration:

... fixed-size-buffer-declaration

#### A.3.9 Arrays

array-initializer: { variable-initializer-list<sub>opt</sub> } { variable-initializer-list , }

variable-initializer-list: variable-initializer variable-initializer-list , variable-initializer

variable-initializer: expression array-initializer

# A.3.10 Interfaces

interface-declaration: attributes<sub>opt</sub> interface-modifiers<sub>opt</sub> partial<sub>opt</sub> interface identifier variant-type-parameter-list<sub>opt</sub> interface-base<sub>opt</sub> type-parameter-constraints-clauses<sub>opt</sub> interface-body ;<sub>opt</sub>

interface-modifiers:

interface-modifier interface-modifiers interface-modifier

```
interface-modifier:
             new
             public
             protected
             internal
             private
        variant-type-parameter-list:
             < variant-type-parameters >
        variant-type-parameters:
             attributes<sub>opt</sub> variance-annotation<sub>opt</sub> type-parameter
             variant-type-parameters , attributes<sub>opt</sub> variance-annotation<sub>opt</sub> type-parameter
        variance-annotation:
             in
             out
        interface-base:
             : interface-type-list
        interface-body:
             { interface-member-declarations<sub>opt</sub> }
        interface-member-declarations:
             interface-member-declaration
             interface-member-declarations interface-member-declaration
        interface-member-declaration:
             interface-method-declaration
             interface-property-declaration
             interface-event-declaration
             interface-indexer-declaration
        interface-method-declaration:
             attributes<sub>opt</sub> new<sub>opt</sub> return-type identifier type-parameter-list<sub>opt</sub>
                      ( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub> ;
        interface-property-declaration:
             attributes<sub>opt</sub> new<sub>opt</sub> type identifier { interface-accessors }
        interface-accessors:
             attributesopt get ;
             attributes<sub>opt</sub> set ;
             attributesopt get ; attributesopt set ;
             attributesopt set ; attributesopt get ;
        interface-event-declaration:
             attributes<sub>opt</sub> new<sub>opt</sub> event type identifier ;
        interface-indexer-declaration:
             attributes<sub>opt</sub> new<sub>opt</sub> type this [ formal-parameter-list ] { interface-accessors }
A.3.11 Enums
        enum-declaration:
             attributes<sub>opt</sub> enum-modifiers<sub>opt</sub> enum identifier enum-base<sub>opt</sub> enum-body ;<sub>opt</sub>
        enum-base:
             : integral-type
```

enum-body:

{ *enum-member-declarations*<sub>opt</sub> }

{ enum-member-declarations , }

enum-modifiers: enum-modifier enum-modifiers enum-modifier

enum-modifier: new public protected internal

private

enum-member-declarations: enum-member-declaration enum-member-declarations , enum-member-declaration

enum-member-declaration: attributes<sub>opt</sub> identifier attributes<sub>opt</sub> identifier = constant-expression

#### A.3.12 Delegates

delegate-declaration: attributes<sub>opt</sub> delegate-modifiers<sub>opt</sub> delegate return-type identifier variant-type-parameter-list<sub>opt</sub> ( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub> ;

delegate-modifiers: delegate-modifier delegate-modifiers delegate-modifier

delegate-modifier:

new public protected internal private

## A.3.13 Attributes

global-attributes: global-attribute-sections

global-attribute-sections: global-attribute-section global-attribute-sections global-attribute-section

global-attribute-section:

- [ global-attribute-target-specifier attribute-list ]
- [ global-attribute-target-specifier attribute-list ,]

global-attribute-target-specifier: global-attribute-target :

global-attribute-target: identifier equal to assembly or module

attributes: attribute-sections

attribute-sections: attribute-section attribute-sections attribute-section

attribute-section: [ attribute-target-specifier<sub>opt</sub> attribute-list ] [ attribute-target-specifier<sub>opt</sub> attribute-list , ] attribute-target-specifier: attribute-target : attribute-target: identifier not equal to assembly or module keyword attribute-list: attribute attribute-list , attribute attribute: attribute-name attribute-argumentsopt attribute-name: type-name attribute-arguments: ( *positional-argument-list*<sub>opt</sub> ) ( positional-argument-list , named-argument-list ) ( *named-argument-list* ) positional-argument-list: positional-argument positional-argument-list , positional-argument *positional-argument:* argument-nameopt attribute-argument-expression named-argument-list: named-argument named-argument-list , named-argument named-argument: *identifier* = attribute-argument-expression attribute-argument-expression: expression A.4 Grammar extensions for unsafe code class-modifier: unsafe struct-modifier: unsafe interface-modifier: unsafe delegate-modifier: ... unsafe field-modifier: ••• unsafe

```
method-modifier:
    ...
    unsafe
property-modifier:
    ...
    unsafe
event-modifier:
    ...
    unsafe
indexer-modifier:
    ...
    unsafe
operator-modifier:
    unsafe
constructor-modifier:
    ...
    unsafe
finalizer-declaration:
    attributes_{opt} extern<sub>opt</sub> unsafe<sub>opt</sub> ~ identifier ( ) finalizer-body
    attributes_{opt} unsafe<sub>opt</sub> extern<sub>opt</sub> ~ identifier ( ) finalizer-body
static-constructor-modifiers:
    extern<sub>opt</sub> unsafe<sub>opt</sub> static
    unsafeopt externopt static
    externopt static unsafeopt
    unsafe<sub>opt</sub> static extern<sub>opt</sub>
    static externopt unsafeopt
    static unsafeopt externopt
embedded-statement:
    ...
    unsafe-statement
unsafe-statement:
    unsafe block
type:
    ...
    pointer-type
non-array-type:
    ...
    pointer-type
pointer-type:
    unmanaged-type *
    void *
unmanaged-type:
    type
primary-no-array-creation-expression:
    pointer-member-access
    pointer-element-access
```

```
unary-expression:
    pointer-indirection-expression
    addressof-expression
pointer-indirection-expression:
    * unary-expression
pointer-member-access:
    primary-expression -> identifier type-argument-list<sub>opt</sub>
pointer-element-access:
    primary-no-array-creation-expression [ expression ]
addressof-expression:
    & unary-expression
embedded-statement:
    fixed-statement
fixed-statement:
    fixed ( pointer-type fixed-pointer-declarators ) embedded-statement
fixed-pointer-declarators:
    fixed-pointer-declarator
    fixed-pointer-declarators , fixed-pointer-declarator
fixed-pointer-declarator:
    identifier = fixed-pointer-initializer
fixed-pointer-initializer:
    & variable-reference
    expression
struct-member-declaration:
    fixed-size-buffer-declaration
fixed-size-buffer-declaration:
    attributes<sub>opt</sub> fixed-size-buffer-modifiers<sub>opt</sub> fixed buffer-element-type
            fixed-size-buffer-declarators ;
fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-size-buffer-modifier fixed-size-buffer-modifiers
fixed-size-buffer-modifier:
    new
    public
    protected
    internal
    private
    unsafe
buffer-element-type:
    type
fixed-size-buffer-declarators:
    fixed-size-buffer-declarator
    fixed-size-buffer-declarator , fixed-size-buffer-declarators
fixed-size-buffer-declarator:
    identifier [ constant-expression ]
```

local-variable-initializer:

... stackalloc-initializer

stackalloc-initializer:
 stackalloc unmanaged-type [ expression ]

End of informative text.

# Annex B. Portability issues

This clause is informative.

# **B.1 General**

This annex collects some information about portability that appears in this specification.

# **B.2 Undefined behavior**

The behavior is undefined in the following circumstances:

- The behavior of the enclosing async function when an awaiter's implementation of the interface methods INotifyCompletion.OnCompleted and ICriticalNotifyCompletion.UnsafeOnCompleted does not cause the resumption delegate to be invoked at most once (§12.8.8.4).
- 2. Passing pointers as ref or out parameters (§23.3).
- 3. When dereferencing the result of converting one pointer type to another and the resulting pointer is not correctly aligned for the pointed-to type. (§23.5.1)
- 4. When the unary \* operator is applied to a pointer containing an invalid value (§23.6.2).
- 5. When a pointer is subscripted to access an out-of-bounds element (§23.6.4).
- 6. Modifying objects of managed type through fixed pointers (§23.7)
- 7. The content of memory newly allocated by stackalloc (§23.9).
- 8. Attempting to allocate a negative number of items using stackalloc (§23.9).

# **B.3 Implementation-defined behavior**

A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

- 1. The behavior when an identifier not in Normalization Form C is encountered (§7.4.3).
- 2. The interpretation of the input-characters in the pp-pragma-text of a #pragma directive (§7.5.9).
- 3. The values of any application parameters passed to Main by the host environment prior to application startup (§8.1).
- 4. The precise structure of the expression tree, as well as the exact process for creating it, when an anonymous function is converted to an expression-tree (§11.7.3).
- 5. Whether a System.ArithmeticException (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand, when in an unchecked context and the left operand of an integer division is the maximum negative int or long value and the right operand is -1 (§12.9.3).
- 6. When a System.ArithmeticException (or a subclass thereof) is thrown when performing a decimal remainder operation (§12.9.4).
- 7. The impact of thread termination when a thread has no handler for an exception, and the thread is itself terminated (§13.10.6).
- 8. 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. (§21.4)

- 9. The mappings between pointers and integers (§23.5.1).
- 10. The effect of applying the unary \* operator to a null pointer (§23.6.2).
- 11. The behavior when pointer arithmetic overflows the domain of the pointer type (§23.6.6, §23.6.7).
- 12. The result of the sizeof operator for non-pre-defined value types (§23.6.9).
- 13. The behavior of the fixed statement if the array expression is null or if the array has zero elements (§23.7).
- 14. The behavior of the fixed statement if the string expression is null (§23.7).
- 15. The value returned when a stack allocation of size zero is made (§23.9).

## **B.4 Unspecified behavior**

- 1. The time at which the finalizer (if any) for an object is run, once that object has become eligible for finalization (§8.9).
- 2. The value of the result when converting out-of-range values from float or double values to an integral type in an unchecked context (§11.3.2).
- 3. The exact target object and target method of the delegate produced from an *anonymous-method-expression* contains (§11.7.2).
- 4. The layout of arrays, except in an unsafe context (§12.7.11.5).
- 5. Whether there is any way to execute the *block* of an anonymous function other than through evaluation and invocation of the *lambda-expression* or *anonymous-method-expression* (§12.16.3).
- 6. The exact timing of static field initialization (§15.5.6.2).
- 7. The result of invoking MoveNext when an enumerator object is running (§15.14.5.2).
- 8. The result of accessing Current when an enumerator object is in the before, running, or after states (§15.14.5.3).
- 9. The result of invoking Dispose when an enumerator object is in the running state (§15.14.5.4).
- 10. The attributes of a type declared in multiple parts are determined by combining, in an unspecified order, the attributes of each of its parts (§22.3).
- 11. The order in which members are packed into a struct (§23.6.9).
- 12. An exception occurs during finalizer execution, and that execution is not caught (§21.4).
- 13. If more than one member matches, which member is the implementation of I.M.(§18.6.5)

# **B.5 Other Issues**

- 1. The exact results of floating-point expression evaluation can vary from one implementation to another, because an implementation is permitted to evaluate such expressions using a greater range and/or precision than is required. (§9.3.7)
- 2. The CLI reserves certain signatures for compatibility with other programming languages. (§15.3.9.7)

#### End of informative text.

# **Annex C. Standard library**

# C.1 General

A conforming C# implementation shall provide a minimum set of types having specific semantics. These types and their members are listed here, in alphabetical order by namespace and type. For a formal definition of these types and their members, refer to ISO/IEC 23271:2012 *Common Language Infrastructure (CLI), Partition IV; Base Class Library (BCL), Extended Numerics Library, and Extended Array Library*, which are included by reference in this specification.

# This text is informative.

The standard library is intended to be the minimum set of types and members required by a conforming C# implementation. As such, it contains only those members that are explicitly required by the C# language specification.

It is expected that a conforming C# implementation will supply a significantly more extensive library that enables useful programs to be written. For example, a conforming implementation might extend this library by

- Adding namespaces.
- Adding types.
- Adding members to non-interface types.
- Adding intervening base classes or interfaces.
- Having struct and class types implement additional interfaces.
- Adding attributes (other than the ConditionalAttribute) to existing types and members.

# End of informative text.

# C.2 Standard Library Types defined in ISO/IEC 23271

```
namespace System
{
   public class ArgumentException : SystemException
      public ArgumentException();
      public ArgumentException(string message);
      public ArgumentException(string message, Exception innerException);
   }
namespace System
{
   public delegate void Action();
}
namespace System
{
   public class ArithmeticException : Exception
      public ArithmeticException();
      public ArithmeticException(string message);
      public ArithmeticException(string message, Exception innerException);
   }
}
```

```
namespace System
{
   public abstract class Array : IList, ICollection, IEnumerable
      public int Length { get; }
      public int Rank { get;
      public int GetLength(int dimension);
   }
}
namespace System
Ł
   public class ArrayTypeMismatchException : Exception
   ł
      public ArrayTypeMismatchException();
      public ArrayTypeMismatchException(string message);
      public ArrayTypeMismatchException(string message,
          Exception innerException);
   }
}
namespace System
Ł
   [AttributeUsageAttribute(AttributeTargets.All, Inherited = true,
      AllowMultiple = false]
   public abstract class Attribute
      protected Attribute();
   }
}
namespace System
{
   public enum AttributeTargets
      Assembly = 0x1,
Module = 0x2,
      Class = 0x4.
      Struct = 0x8,
      Enum = 0x10,
      Constructor = 0x20,
      Method = 0x40
      Property = 0 \times 80,
      Field = 0x100,
      Event = 0x200,
      Interface = 0x400,
      Parameter = 0x800,
      Delegate = 0x1000
      ReturnValue = 0x2000,
      GenericParameter = 0x4000,
      All = 0x7FFF
   }
}
namespace System
{
   [AttributeUsageAttribute(AttributeTargets.Class, Inherited = true)]
   public sealed class AttributeUsageAttribute : Attribute
      public AttributeUsageAttribute(AttributeTargets validOn);
      public bool AllowMultiple { get; set; }
public bool Inherited { get; set; }
public AttributeTargets ValidOn { get; }
   }
}
```

```
namespace System
{
   public struct Boolean
}
namespace System
{
   public struct Byte
}
namespace System
{
   public struct Char
}
namespace System
Ł
   public struct Decimal
   }
}
namespace System
{
   public abstract class Delegate
}
namespace System
{
   public class DivideByZeroException : ArithmeticException
   {
      public DivideByZeroException();
      public DivideByZeroException(string message);
      public DivideByZeroException(string message, Exception innerException);
   }
}
namespace System
{
   public struct Double
}
namespace System
{
   public abstract class Enum : ValueType
      protected Enum();
   }
}
```

```
namespace System
{
   public class Exception
      public Exception();
      public Exception(string message);
      public Exception(string message, Exception innerException);
      public sealed Exception InnerException { get; }
      public virtual string Message { get; }
   }
}
namespace System
Ł
   public class GC
   }
}
namespace System
{
   public interface IDisposable
      public void Dispose();
   }
}
namespace System
{
   public sealed class IndexOutOfRangeException : Exception
   ł
      public IndexOutOfRangeException();
      public IndexOutOfRangeException(string message);
      public IndexOutOfRangeException(string message,
         Exception innerException);
   }
}
namespace System
{
   public struct Int16
   3
}
namespace System
Ł
   public struct Int32
}
namespace System
{
   public struct Int64
}
namespace System
Ł
   public struct IntPtr
   }
}
```

```
namespace System.Runtime.CompilerServices
{
   public sealed class IndexerNameAttribute: Attribute
      public IndexerNameAttribute(String indexerName);
   }
}
namespace System.Collections.Generic
   public interface IReadOnlyCollection<out T> : IEnumerable<T>
      int Count { get; }
   }
}
namespace System.Collections.Generic
ł
   public interface IReadOnlyList<out T> : IReadOnlyCollection<T>
      T this[int index] { get; }
   }
}
namespace System
ł
   public class InvalidCastException : Exception
      public InvalidCastException();
      public InvalidCastException(string message);
      public InvalidCastException(string message, Exception innerException);
   }
}
namespace System
ł
   public class InvalidOperationException : Exception
      public InvalidOperationException();
      public InvalidOperationException(string message);
public InvalidOperationException(string message,
         Exception innerException);
   }
}
namespace System.Reflection
Ł
   public abstract class MemberInfo
      protected MemberInfo();
   }
}
namespace System
ł
   public class NotSupportedException : Exception
      public NotSupportedException();
      public NotSupportedException(string message);
      public NotSupportedException(string message, Exception innerException);
   }
}
```

```
namespace System
{
   public struct Nullable<T>
      public bool HasValue { get; }
      public T Value { get; }
   }
}
namespace System
ł
   public class NullReferenceException : Exception
      public NullReferenceException();
      public NullReferenceException(string message);
      public NullReferenceException(string message, Exception innerException);
   }
}
namespace System
{
   public class Object
   ł
      public Object();
      ~Object();
      public virtual bool Equals(object obj);
      public virtual int GetHashCode();
public Type GetType();
      public virtual string ToString();
   }
}
namespace System
Ł
   [AttributeUsageAttribute(AttributeTargets.Class
            AttributeTargets.Struct
            AttributeTargets.Enum | AttributeTargets.Interface
            AttributeTargets.Constructor | AttributeTargets.Method
AttributeTargets.Property | AttributeTargets.Field
           AttributeTargets.Event | AttributeTargets.Delegate,
Inherited = false)]
   public sealed class ObsoleteAttribute : Attribute
      public ObsoleteAttribute();
      public ObsoleteAttribute(string message);
      public ObsoleteAttribute(string message, bool error);
      public bool IsError { get; }
      public string Message { get; }
   }
}
namespace System
Ł
   public class OutOfMemoryException : Exception
      public OutOfMemoryException();
      public OutOfMemoryException(string message);
      public OutOfMemoryException(string message, Exception innerException);
   }
}
```

```
namespace System
{
   public class OverflowException : ArithmeticException
      public OverflowException();
      public OverflowException(string message);
      public OverflowException(string message, Exception innerException);
   }
}
namespace System
Ł
   public struct SByte
}
namespace System
{
   public struct Single
}
namespace System
Ł
   public sealed class StackOverflowException : Exception
      public StackOverflowException();
      public StackOverflowException(string message);
      public StackOverflowException(string message, Exception innerException);
   }
}
namespace System
{
   public sealed class String : IEnumerable<Char>, IEnumerable
      public int Length { get; }
      public char this[int index] { get; }
   }
}
namespace System
{
   public abstract class Type : MemberInfo
}
namespace System
{
   public sealed class TypeInitializationException : Exception
   ł
      public TypeInitializationException(string fullTypeName,
         Exception innerException);
   }
}
namespace System
ł
   public struct UInt16
   }
}
```

```
namespace System
{
   public struct UInt32
}
namespace System
{
   public struct UInt64
}
namespace System
{
   public struct UIntPtr
}
namespace System
Ł
   public abstract class ValueType
       protected ValueType();
   }
}
namespace System.Collections
{
   public interface ICollection : IEnumerable
       public int Count { get; }
public bool IsSynchronized { get; }
       public object SyncRoot { get; }
public void CopyTo(Array array, int index);
   }
}
namespace System.Collections
{
   public interface IEnumerable
       public IEnumerator GetEnumerator();
   }
}
namespace System.Collections
{
   public interface IEnumerator
       public object Current { get; }
       public bool MoveNext();
public void Reset();
   }
}
namespace System.Collections
Ł
   public interface IList : ICollection, IEnumerable
    ł
       public bool IsFixedSize { get; }
public bool IsReadOnly { get; }
public object this[int index] { get; set; }
       public int Add(object value);
       public void Clear();
public bool Contains(object value);
```

```
public int IndexOf(object value);
      public void Insert(int index, object value);
      public void Remove(object value);
public void RemoveAt(int index);
   }
}
namespace System.Collections.Generic
Ł
   public interface ICollection<T> : IEnumerable<T>
      public int Count { get; }
public bool IsReadOnly { get; }
public void Add(T item);
public void Clear();
      public bool Contains(T item);
      public void CopyTo(T[] array, int arrayIndex);
      public bool Remove(T item);
   }
}
namespace System.Collections.Generic
ł
   public interface IEnumerable<T> : IEnumerable
      public IEnumerator<T> GetEnumerator();
   }
}
namespace System.Collections.Generic
ł
   public interface IEnumerator<T> : IDisposable, IEnumerator
      public T Current { get; }
   }
}
namespace System.Collections.Generic
{
   public interface IList<T> : ICollection<T>
      public T this[int index] { get; set; }
public int IndexOf(T item);
      public void Insert(int index, T item);
      public void RemoveAt(int index);
   }
}
namespace System.Diagnostics
4
   [AttributeUsageAttribute(AttributeTargets.Method
    AttributeTargets.Class, AllowMultiple = true)]
   public sealed class ConditionalAttribute : Attribute
       public ConditionalAttribute(string conditionString);
      public string ConditionString { get; }
   }
}
namespace System. Threading
Ł
   public static class Monitor
   ł
       public static void Enter(object obj);
      public static void Exit(object obj);
   }
}
```

# C.3 Standard Library Types not defined in ISO/IEC 23271:2012

The following types, including the members listed, must be defined in a conforming standard library. (These types might be defined in a future edition of ISO/IEC 23271.) It is expected that many of these types will have more members available than are listed.

A conforming implementation may provide Task.GetAwaiter() and Task<T>.GetAwaiter() as extension methods.

```
namespace System.Runtime.CompilerServices
{
   [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
   public sealed class CallerFilePathAttribute : Attribute
   {
      public CallerFilePathAttribute() {}
   }
}
namespace System.Runtime.CompilerServices
ł
   [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
   public sealed class CallerLineNumberAttribute : Attribute
   ł
      public CallerLineNumberAttribute() {}
   }
}
namespace System.Runtime.CompilerServices
   [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
   public sealed class CallerMemberNameAttribute : Attribute
   {
      public CallerMemberNameAttribute() {}
   }
}
namespace System.Linq.Expressions
   public sealed class Expression<TDelegate>
   {
      // See Section 12.7.3 for details on what
      // Delegate types (TDelegate) must be supported,
      // and which may be omitted.
      public TDelegate Compile();
   }
}
namespace System.Runtime.CompilerServices
   public interface INotifyCompletion
   {
      void OnCompleted(Action continuation);
   }
}
namespace System.Runtime.CompilerServices
   public interface ICriticalNotifyCompletion : INotifyCompletion
```

```
{
     void UnsafeOnCompleted(Action continuation);
  }
}
namespace System.Threading.Tasks
{
  public class Task
  {
     public System.Runtime.CompilerServices.TaskAwaiter GetAwaiter();
  }
}
namespace System.Threading.Tasks
  public class Task<TResult> : System.Threading.Tasks.Task
  {
     public new System.Runtime.CompilerServices.TaskAwaiter<T>
       GetAwaiter();
  }
}
namespace System.Runtime.CompilerServices
{
  {
     public bool IsCompleted { get; }
     public void GetResult();
  }
}
namespace System.Runtime.CompilerServices
{
  public struct TaskAwaiter<T> : ICriticalNotifyCompletion,
INotifyCompletion
  {
     public bool IsCompleted { get; }
     public T GetResult();
  }
}
```

# **Annex D. Documentation comments**

# This annex is informative.

# **D.1 General**

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. In source code files, comments having a certain form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Comments using such syntax are called *documentation comments*. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method). The XML generation tool is called the *documentation generator*. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the *documentation file*. A documentation file is used as input to a *documentation viewer*; a tool intended to produce some sort of visual display of type information and its associated documentation.

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, but use of these tags is not required, and other tags may be used if desired, as long the rules of well-formed XML are followed. 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.

# **D.2 Introduction**

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are single-line comments that start with three slashes (///), or delimited comments that start with a slash and two stars (/\*\*). They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate. Attribute sections (§22.3) are considered part of declarations, so documentation comments must precede attributes applied to a type or member.

# Syntax:

single-line-doc-comment:: /// input-characters<sub>opt</sub> delimited-doc-comment:: /\*\* delimited-comment-text<sub>opt</sub> \*/

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 can include *whitespace* characters after, as well as before, the *asterisk* character.

# Example:

## ECMA-334

/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
 /// <summary>method <c>draw</c> renders the point.</summary>
 void draw() {...}
}

The text within documentation comments must be well formed according to the rules of XML (http://www.w3.org/TR/REC-xml). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in §D.3. Some of the recommended tags have special meanings:

- The <param> tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The cref attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a cref attribute, the documentation generator must respect namespace visibility according to using statements appearing within the source code. For code elements that are generic, the normal generic syntax (e.g.; "List<T>") cannot be used because it produces invalid XML. Braces can be used instead of brackets (e.g.; "List{T}"), or the XML escape syntax can be used (e.g.; "List&lt;T&gt;").
- The <summary> tag is intended to be used by a documentation viewer to display additional information about a type or member.
- The <include> tag includes information from an external XML file.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the type or member.

# **D.3 Recommended tags**

# D.3.1 General

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

Тад	Reference	Purpose
<c></c>	§D.3.2	Set text in a code-like font
<code></code>	§D.3.3	Set one or more lines of source code or program output
<example></example>	§D.3.4	Indicate an example
<exception></exception>	§D.3.5	Identifies the exceptions a method can throw
<list></list>	§D.3.6	Create a list or table
<include></include>	§D.3.6	Includes XML from an external file
<para></para>	§D.3.8	Permit structure to be added to text
<param/>	§D.3.9	Describe a parameter for a method or constructor
<paramref></paramref>	§D.3.10	Identify that a word is a parameter name
<permission></permission>	§D.3.11	Document the security accessibility of a member
<remarks></remarks>	§D.3.12	Describe additional information about a type
<returns></returns>	§D.3.13	Describe the return value of a method
<see></see>	§D.3.14	Specify a link
<seealso></seealso>	§D.3.15	Generate a See Also entry
<summary></summary>	§D.3.16	Describe a type or a member of a type
<typeparam></typeparam>	§D.3.17	Describe a type parameter for a generic type or method
<typeparamref></typeparamref>	§D.3.18	Identify that a word is a type parameter name
<value></value>	§D.3.17	Describe a property

# D.3.2 <c>

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use <code> (§D.3.3).

#### Syntax:

```
<c>text</c>
```

## Example:

/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
public class Point
{
 // ...
}

# D.3.3 <code>

This tag is used to set one or more lines of source code or program output in some special font. For small code fragments in narrative, use  $\langle c \rangle$  (§D.3.2).

# Syntax:

```
<code>source code or program output</code>
```

Example:

```
/// <summary>This method changes the point's location by
/// the given x- and y-offsets.
/// <example>For example:
/// code>
/// Point p = new Point(3,5);
/// p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

# D.3.4 <example>

This tag allows example code within a comment, to specify how a method or other library member might be used. Ordinarily, this would also involve use of the tag <code> (§D.3.3) as well.

#### Syntax:

<example>*description*</example>

#### Example:

See <code> (§D.3.3) for an example.

## D.3.5 <exception>

This tag provides a way to document the exceptions a method can throw.

#### Syntax:

```
<exception cref="member">description</exception>
```

#### where

cref="member"

The name of a member. The documentation generator checks that the given member exists and translates *member* to the canonical element name in the documentation file.

```
description
```

A description of the circumstances in which the exception is thrown.

#### Example:

```
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}
```

# D.3.6 <include>

This tag allows including information from an XML document that is external to the source code file. The external file must be a well-formed XML document, and an XPath expression is applied to that document to specify what XML from that document to include. The <include> tag is then replaced with the selected XML from the external document.

Syntax:

<include file="filename" path="xpath" />

where

file="filename"

The file name of an external XML file. The file name is interpreted relative to the file that contains the include tag.

path="xpath"

An XPath expression that selects some of the XML in the external XML file.

#### Example:

If the source code contained a declaration like:

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

and the external file "docs.xml" had the following contents:

```
<?xml version="1.0"?>
<extradoc>
<class name="IntList">
<summary>
Contains a list of integers.
</summary>
</class>
<class name="StringList">
<summary>
Contains a list of integers.
</summary>
</class>
</extradoc>
```

then the same documentation is output as if the source code contained:

```
/// <summary>
/// Contains a list of integers.
/// </summary>
public class IntList { ... }
```

#### D.3.7 <list>

This tag is used to create a list or table of items. It can contain a <listheader> block to define the heading row of either a table or definition list. (When defining a table, only an entry for *term* in the heading need be supplied.)

Each item in the list is specified with an <item> block. When creating a definition list, both *term* and *description* must be specified. However, for a table, bulleted list, or numbered list, only *description* need be specified.

#### Syntax:

```
<list type="bullet" | "number" | "table">
<listheader>
<term>term</term>
<description>description</description>
</listheader>
<item>
<term>term</term>
<description>description</description>
</item>
...
```

```
<item>
<term>term</term>
<description>description</description>
</item>
</list>
```

where

term

The term to define, whose definition is in *description*.

#### description

Either an item in a bullet or numbered list, or the definition of a *term*.

#### Example:

```
public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <list type="bullet">
    /// <liem>
    /// <description>Item 1.</description>
    /// </liem>
    /// <litem>
    /// <litem>
    /// <description>Item 2.</description>
    /// </list>
    /// </list>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}
```

# D.3.8 <para>

This tag is for use inside other tags, such as <summary> (§D.3.16) or <returns> (§D.3.13), and permits structure to be added to text.

#### Syntax:

<para>content</para>

where

content

The text of the paragraph.

#### Example:

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trvial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

# D.3.9 <param>

This tag is used to describe a parameter for a method, constructor, or indexer.

#### Syntax:

```
<param name="name">description</param>
```

where

name

The name of the parameter.

#### description

A description of the parameter.

#### Example:

```
/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

# D.3.10 <paramref>

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

#### Syntax:

<paramref name="name"/>

#### where

name

The name of the parameter.

#### Example:

```
/// <summary>This constructor initializes the new Point to
/// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

# D.3.11 <permission>

This tag allows the security accessibility of a member to be documented.

#### Syntax:

```
<permission cref="member">description</permission>
```

#### where

member

The name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.

#### description

A description of the access to the member.

#### Example:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>
public static void Test() {
    // ...
}
```

# D.3.12 <remarks>

This tag is used to specify extra information about a type. Use <summary> (§D.3.16) to describe the type itself and the members of a type.

#### Syntax:

```
<remarks>description</remarks>
```

#### where

description

The text of the remark.

#### Example:

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remarks>Uses polar coordinates</remarks>
public class Point
{
    // ...
}
```

## D.3.13 <returns>

This tag is used to describe the return value of a method.

#### Syntax:

<returns>*description*</returns>

where

description

A description of the return value.

#### Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
/// without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
   return "(" + X + "," + Y + ")";
}
```

# D.3.14 <see>

This tag allows a link to be specified within text. Use <seealso> (§D.3.15) to indicate text that is to appear in a *See Also* subclause.

#### Syntax:

<see cref="member"/>

where

member

The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

#### Example:

```
/// <summary>This method changes the point's location to
/// the given coordinates. <see cref="Translate"/></summary>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

```
/// <summary>This method changes the point's location by
/// the given x- and y-offsets. <see cref="Move"/>
/// </summary>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

# D.3.15 <seealso>

This tag allows an entry to be generated for the *See Also* subclause. Use <see> (§D.3.14) to specify a link from within text.

Syntax:

<seealso cref="member"/>

where

member

The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method determines whether two Points have the same
/// location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

# D.3.16 <summary>

This tag can be used to describe a type or a member of a type. Use <remarks> (§D.3.12) to describe the type itself.

#### Syntax:

<summary>*description*</summary>

where

description

A summary of the type or member.

#### Example:

```
/// <summary>This constructor initializes the new Point to
(0,0).</summary>
public Point() : this(0,0) {
```

# D.3.17 <typeparam>

This tag is used to describe a type parameter for a generic type or method.

Syntax:

```
<typeparam name="name">description</typeparam>
```

where

name

The name of the type parameter.

description

A description of the typeparameter.

#### Example:

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

# D.3.18 <typeparamref>

This tag is used to indicate that a word is a type parameter. The documentation file can be processed to format this type parameter in some distinct way.

#### Syntax:

```
<typeparamref name="name"/>
```

where

name

The name of the type parameter.

#### Example:

```
/// <summary>This method fetches data and returns a list of <typeparamref
name="T"> "/>"> .</summary>
/// <param name="string">query to execute</param>
public List<T> FetchData<T>(string query) {
    ""
```

# D.3.19 <value>

This tag allows a property to be described.

#### Syntax:

<value>property description</value>

where

property description

A description for the property.

#### Example:

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
```

```
get { return x; }
set { x = value; }
```

# D.4 Processing the documentation file

# **D.4.1 General**

The following information is intended for C# implementations targeting the CLI.

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

# D.4.2 ID string format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined:

Character	Description
E	Event
F	Field
М	Method (including constructors, finalizers, and operators)
N	Namespace
Р	Property (including indexers)
Т	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by # (U+0023) characters. (It is assumed that no element has this character in its name.)
- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows:
  - Arguments are represented by their documentation name, which is based on their fully qualified name, modified as follows:
    - Arguments that represent generic types have an appended "" character followed by the number of type parameters
    - Arguments having the out or ref modifier have an @ following their type name. Arguments passed by value or via params have no special notation.
    - Arguments that are arrays are represented as [ *lowerbound* : *size* , ... , *lowerbound* : *size* ] where the number of commas is the rank less one, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the ":" is omitted as well. Jagged arrays are represented by one "[]" per level.
    - Arguments that have pointer types other than void are represented using a \* following the type name. A void pointer is represented using a type name of System.Void.
    - Arguments that refer to generic type parameters defined on types are encoded using the "`" character followed by the zero-based index of the type parameter.
    - Arguments that use generic type parameters defined in methods use a double-backtick "``" instead of the "`" used for types.
    - Arguments that refer to constructed generic types are encoded using the generic type, followed by "{", followed by a comma-separated list of type arguments, followed by "}".

# D.4.3 ID string examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

• Types are represented using their fully qualified name, augmented with generic information:

```
enum Color { Red, Blue, Green }
   namespace Acme
   {
      interface IProcess { ... }
      struct ValueType { ... }
      class Widget: IProcess
          public class NestedClass { ... }
          public interface IMenuItem { ... }
          public delegate void Del(int i);
          public enum Direction { North, South, East, West }
      }
      class MyList<T>
          class Helper<U,V>{ ... }
      }
   }
   "T:Color"
   "T:Acme.IProcess"
   "T:Acme.ValueType"
   "T:Acme.Widget
   "T:Acme.Widget.NestedClass"
   "T:Acme.Widget.IMenuItem"
   "T:Acme.Widget.Del"
   "T:Acme.Widget.Direction"
   "T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
Fields are represented by their fully qualified name.
   namespace Acme
```

```
{
   struct ValueType
   {
       private int total;
   }
   class Widget: IProcess
       public class NestedClass
       {
          private int value;
       }
       private string message;
       private static Color defaultColor;
       private const double PI = 3.14159;
       protected readonly double monthlyAverage;
      private long[] array1;
private Widget[,] array2;
private unsafe int *pCount;
       private unsafe float **ppválues;
   }
}
```

•

```
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

```
Constructors.
•
```

```
namespace Acme
{
   class Widget: IProcess
      static Widget() { ... }
      public Widget() { ... }
      public Widget(string s) { ... }
   }
}
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

Finalizers. •

```
namespace Acme
{
   class Widget: IProcess
      ~Widget() { ... }
   ł
}
```

```
"M:Acme.Widget.Finalize"
```

```
Methods.
•
```

{

```
namespace Acme
       struct ValueType
       {
              public void M(int i) { ... }
       }
       class Widget: IProcess
              public class NestedClass
              {
                     public void M(int i) { ... }
              }
             public static void MO() { ... }
public void M1(char c, out float f, ref ValueType v) { ... }
public void M2(short[] x1, int[,] x2, long[][] x3) { ... }
public void M3(long[][] x3, Widget[][,,] x4) { ... }
public unsafe void M4(char *pc, Color **pf) { ... }
public unsafe void M5(void *pv, double *[][,] pd) { ... }
public void M6(int i, params object[] args) { ... }
      class MyList<T>
{
              public void Test(T t) { ... }
       }
```

.

```
class UseList
             public void Process(MyList<int> list) { ... }
            public MyList<T> GetValues<T>(T value) { ... } }
    }
    "M:Acme.ValueType.M(System.Int32)"
    "M:Acme.Widget.NestedClass.M(System.Int32)"
    "M:Acme.Widget.MO
    "M:Acme.Widget.M0
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
    "MAcme.Widget.M3(System.Intof[][],Acme.Widget[0:,0:,0:]
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
    "M:Acme.UseList.Process(Acme.MyList{System.Int32})"
    "M:Acme.UseList.GetValues``1(``0)
Properties and indexers.
    namespace Acme
    {
        class Widget: IProcess
        ł
            public int Width {get { ... } set { ... }}
public int this[int i] {get { ... } set { ... }}
public int this[string s, int i] {get { ... } set { ... }}
        }
    }
    "P:Acme.Widget.Width"
    "P:Acme.Widget.Item(System.Int32)"
    "P:Acme.Widget.Item(System.String,System.Int32)"
Events
    namespace Acme
    ł
        class Widget: IProcess
        {
             public event Del AnEvent;
        }
    }
    "E:Acme.Widget.AnEvent"
Unary operators.
    namespace Acme
    Ł
        class Widget: IProcess
        {
             public static widget operator+(widget x) { ... }
        }
    }
    "M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
    The complete set of unary operator function names used is as follows: op_UnaryPlus,
    op_UnaryNegation, op_LogicalNot, op_OnesComplement, op_Increment,
    op_Decrement, op_True, and op_False.
Binary operators.
    namespace Acme
    Ł
        class Widget: IProcess
        Ł
             public static Widget operator+(Widget x1, Widget x2) { ... }
        }
    }
```

"M:Acme.Widget.op\_Addition(Acme.Widget,Acme.Widget)"

The complete set of binary operator function names used is as follows: op\_Addition, op\_Subtraction, op\_Multiply, op\_Division, op\_Modulus, op\_BitwiseAnd, op\_BitwiseOr, op\_ExclusiveOr, op\_LeftShift, op\_RightShift, op\_Equality, op\_Inequality, op\_LessThan, op\_LessThanOrEqual, op\_GreaterThan, and op\_GreaterThanOrEqual.

• Conversion operators have a trailing "~" followed by the return type.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) { ... }
        public static implicit operator long(Widget x) { ... }
    }
}
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

# D.5 An example

#### D.5.1 C# source code

The following example shows the source code of a Point class:

```
namespace Graphics
ł
/// <summary>Class <c>Point</c> models a point in a two-dimensional
plane.
/// </summary>
public class Point
   /// <summary>Instance variable <c>x</c> represents the point's
/// x-coordinate.</summary>
   private int x;
   /// <summary>Instance variable <c>y</c> represents the point's
/// y-coordinate.</summary>
   private int y;
    /// <value>Property <c>X</c> represents the point's x-
coordinate.</value>
   public int X
    ł
       get { return x; }
set { x = value; }
    }
   /// <value>Property <c>Y</c> represents the point's y-
coordinate.</value>
   public int Y
{
       get { return_y; }
       set { y = value; }
   }
   /// <summary>This constructor initializes the new Point to
/// (0,0).</summary>
public Point() : this(0,0) {}
```

```
/// <summary>This constructor initializes the new Point to
          (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
   /// <param><c>xor</c> is the new Point's x-coordinate.</param>
/// <param><c>yor</c> is the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
      X = xor;
       Y = yor;
   }
   /// <summary>This method changes the point's location to
/// the given coordinates. <see cref="Translate"/></summary>
   /// <param><c>xor</c> is the new x-coordinate.</param>
   /// <param><c>yor</c> is the new y-coordinate </param>
   public void Move(int xor, int yor) {
       X = xor;
       Y = yor;
   }
   /// <summary>This method changes the point's location by
           the given x- and y-offsets.
    ///   the given x- and y-
/// <example>For example:
    /// <code>
    |||
||||
           Point p = new Point(3,5);
           p.Translate(-1,3);
    /// </code>
    /// results in <c>p</c>'s having the value (2,8).
/// <see cref="Move"/></example>
   /// </summary>
   /// <param><c>xor</c> is the relative x-offset.</param>
/// <param><c>yor</c> is the relative y-offset.</param>
   public void Translate(int xor, int yor) {
       X += xor;
       Y += yor;
   }
   /// <summary>This method determines whether two Points have the same
/// location.</summary>
          location.</summary>
   /// < param> < c> o</c> is the object to be compared to the current
object.
   /// </param>
/// <returns>True if the Points have the same location and they have
   ///
        the exact same type; otherwise, false.</returns>
    /// <seealso cref="operator=="/>
    /// <seealso cref="operator!="/>
   public override bool Equals(object o) {
       if (o == null) {
           return false;
       }
       if (this == o) {
           return true;
       }
       if (GetType() == o.GetType()) {
           Point p = (Point)o;
           return (X = p.X) \&\& (Y = p.Y);
       return false;
   }
   /// <summary>Report a point's location as a string.</summary>
   /// <returns>A string representing a point's location, in the form
(x,y),
///
          without any leading, training, or embedded whitespace.</returns>
   public override string ToString() {
    return "(" + X + "," + Y + ")";
```

```
/// <summary>This operator determines whether two Points have the same
            location.</summary>
/// <param><c>pl</c> is the first Point to be compared.</param>
/// <param><c>p2</c> is the second Point to be compared.</param>
/// <returns>True if the Points have the same location and they have
/// the exact same type; otherwise, false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator!="/>
/// <seealso Cref="operator!="/>
public static bool operator==(Point p1, Point p2) {
    if ((object)p1 == null || (object)p2 == null) {
           return false;
     }
      if (p1.GetType() == p2.GetType()) {
            return (p1.X == p2.X) && (p1.Y == p2.Y);
     }
     return false;
}
/// <summary>This operator determines whether two Points have the same
            location.</summary>
/// <param><c>pl</c> is the first Point to be compared.</param>
/// <param><c>p2</c> is the second Point to be compared.</param>
/// <returns>True if the Points do not have the same location and the
/// exact same type; otherwise, false.</returns>
/// exact same type; otherwise, false.</returns>
/// <seealso cref="Equals"/>
public static bool operator!=(Point p1, Point p2) {
     return !(p1 == p2);
}
```

### D.5.2 Resulting XML

}

Here is the output produced by one documentation generator when given the source code for class Point, shown above:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-
dimensional
            plane.
            </summary>
        </member>
        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>
        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>
        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>
```

<member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)"> <summary>This constructor initializes the new Point to </member> <member name="M:Graphics.Point.Move(System.Int32,System.Int32)"> <summary>This method changes the point's location to the given coordinates. <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/></summary> <param><c>xor</c> is the new x-coordinate.</param> <param><c>yor</c> is the new y-coordinate.</param> </member> <member name="M:Graphics.Point.Translate(System.Int32,System.Int32)"> <summary>This method changes the point's location by the given x- and y-offsets. <example>For example: <code> Point p = new Point(3,5);p.Translate(-1,3); </code> results in  $\langle c \rangle p \langle c \rangle$ 's having the value (2,8). </example> <see <param><c>yor</c> is the relative y-offset.</param> </member> <member name="M:Graphics.Point.Equals(System.Object)"> <summary>This method determines whether two Points have the same location.</summary> <param><c>o</c> is the object to be compared to the current object. </param> <returns>True if the Points have the same location and they have the exact same type; otherwise, false.</returns> <seealso cref="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"/> <seealso cref="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"/> </member> <member name="M:Graphics.Point.ToString"> <summary>Report a point's location as a string.</summary>
<returns>A string representing a point's location, in the form (x,y), without any leading, training, or embedded whitespace.</returns> </member>

<member

name="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"> <summary>This operator determines whether two Points have the same location.</summary> <param><c>p1</c> is the first Point to be compared.</param> <param><c>p2</c> is the second Point to be compared.</param>
<returns>True if the Points have the same location and they have the exact same type; otherwise, false.</returns>
<seealso cref="M:Graphics.Point.Equals(System.Object)"/> <seealso cref="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"/> </member> <member name="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"> <summary>This operator determines whether two Points have the same location.</summary> c>p1</c> is the first Point to be compared./param><c>p2</c> is the second Point to be compared./param> <returns>True if the Points do not have the same location and the exact same type; otherwise, false.</returns>
<seealso cref="M:Graphics.Point.Equals(System.Object)"/> <seealso cref="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"/> </member> <member name="M:Graphics.Point.Main"> <summary>This is the entry point of the Point class testing program. <para>This program tests each method and operator, and is intended to be run after any non-trvial maintenance has been performed on the Point class. </member> <member name="P:Graphics.Point.X"> <value>Property <c>X</c> represents the point's x-coordinate.</value> </member> <member name="P:Graphics.Point.Y"> <value>Property <c>Y</c> represents the point's y-coordinate.</value> </member> </members> </doc>

End of informative text.

# Annex E. Bibliography

#### This annex is informative.

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

ISO 31-0:1992, Annex B (informative), *Guide to the rounding of numbers* (This document defines "banker's rounding.")

ISO/IEC 9075-1:2011, Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)

ISO/IEC 9899:2011, Programming languages — C.

ISO/IEC 14882:2011 Programming languages — C++

End of informative text.