*V#*

language specification

Version *1.0*

*05/06/2016*

# preface

V

# (pronounced “Vee Sharp”) is a programming language is an [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming), [imperative](https://en.wikipedia.org/wiki/Imperative_programming), [multi-paradigm](https://en.wikipedia.org/wiki/Multi-paradigm_programming_language) [system programming language](https://en.wikipedia.org/wiki/System_programming_language) created by [Idadi](https://en.wikipedia.org/wiki/Walter_Bright) Arsslen the founder of Phexon Technologies (formerly Arsslensoft) and released in 2016. Though it originated as a re-engineering of C#, V# is a distinct language, having redesigned some core C# features while also taking inspiration from other languages, notably [Java](https://en.wikipedia.org/wiki/Java_(programming_language)), [Python](https://en.wikipedia.org/wiki/Python_(programming_language)), [Ruby](https://en.wikipedia.org/wiki/Ruby_(programming_language)), [C++,](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)) D, and [Eiffel](https://en.wikipedia.org/wiki/Eiffel_(programming_language)).

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

V# was designed *to combine the performance and safety of*[*compiled languages*](https://en.wikipedia.org/wiki/Compiled_language)*with the*[*expressive power*](https://en.wikipedia.org/wiki/Expressive_power_(computer_science))*of modern* [*dynamic languages*](https://en.wikipedia.org/wiki/Dynamic_programming_language)*. Idiomatic V# code is commonly as fast as equivalent C++ and D code, while being shorter and*[*memory-safe*](https://en.wikipedia.org/wiki/Memory_safety)*.*

# TABLE OF CONTENTS

# 

1

# Introduction

**T**he V# programming language is a general purpose systems programming language. To that end, a V# program is a collection of modules that can be compiled separately to native code that is combined with libraries and compiled C, C++, Assembly and other native codes by a linker to create a native executable.

## What is a COMPILER?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in.

A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

* + Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
  + The compiler can spot some obvious programming mistakes.
  + Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

## PHASES OF COMPILATION

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases with well-deﬁned interfaces. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the ﬁrst) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate module. Some of these modules are written by hand, while others may be generated from speciﬁcations. Often, some of the modules can be shared between several compilers. A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

## Lexical Analysis

This is the initial part of reading and analyzing the program text: The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

## Syntax Analysis

This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

## Type checking

This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that does not make sense given the type of the variable, such as trying to use a Boolean value as a function pointer.

## Intermediate code generation

The program is translated to a simple machine independent intermediate language.

Register allocation The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

## Register allocation

The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

## Machine code generation

The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

## Assembling & Linking

The assembly-language code is translated into binary representation and addresses of variables, functions, etc., are determined.

The first three phases are collectively called the frontend of the compiler and the last three phases are collectively called the backend. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimizations and transformations on the intermediate code. Each phase, through checking and transformation, establishes stronger invariants on the things it passes on to the next, so that writing each subsequent phase is easier than if these have to take all the preceding into account. For example, the type checker can assume absence of syntax errors and the code generation can assume absence of type errors. Assembly and linking are typically done by programs supplied by the machine or operating system vendor, and are hence not part of the compiler itself, so we will not further discuss these phases in this specification.

2

# LEXICAL structure

**T**he word “lexical” in the traditional sense means “pertaining to words”. In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens.

A lexical analyzer, or lexer for short, will as its input take a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., lay-out characters (spaces, newlines etc.) and comments.

The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

* Efficiency: A lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non-linear factor involved which may make a separated system smaller than a combined system.
* Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.
* Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

It is usually not terribly difficult to write a lexer by hand: You first read past initial white-space, then you, in sequence, test to see if the next token is a keyword, a number, a variable or whatnot. However, this is not a very good way of handling the problem: You may read the same part of the input repeatedly while testing each possible token and in some cases it may not be clear where the next token ends. Furthermore, a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by lexer generators, which transform human-readable specifications of tokens and white-space into efficient programs.

We will see the same general strategy in the chapter about syntax analysis: Specifications in a well-defined human-readable notation are transformed into efficient programs.

For lexical analysis, specifications are traditionally written using regular expressions: An algebraic notation for describing sets of strings. The generated lexers are in a class of extremely simple programs called finite automata.

This chapter will describe regular expressions and finite automata, their properties and how regular expressions can be converted to finite automata. Finally, we discuss some practical aspects of lexer generators.

## SOURCE TEXT

Programs are written using the Unicode character set. Information about this character set and its associated character encodings may be found at http://[www.unicode.org/](http://www.unicode.org/).

V# source text can be in one of the following formats:

* ASCII
* UTF-8
* Extended ASCII
* ISO 8859
* UTF-16BE
* UTF-16LE
* UCS-2, UCS-4

UTF-8 is a superset of traditional 7-bit ASCII. One of the following UTF BOMs (Byte Order Marks) can be present at the beginning of the source text:

## UTF Byte Order Marks

Format BOM

UTF-8 EF BB BF

UTF-16BE FE FF

UTF-16LE FF FE

ASCII no BOM

If the source ﬁle does not start with a BOM, then the ﬁrst character must be less than or equal to U+0000007F. There are no digraphs or tri graphs in D. The source text is decoded from its source representation into Unicode Characters. The Characters are further divided into: Whitespace, EndOfLine, Comments, Special Token Sequences, Tokens, all followed by EndOfFile. The source text is split into tokens using the maximal munch technique, i.e., the lexical analyzer tries to make the longest token it can. For example, >> is a right shift token, not two greater than tokens. There are two exceptions to this rule:

• A... embedded inside what looks like two ﬂoating point literals, as in 1..2, is interpreted as if the .. was separated by a space from the ﬁrst integer. • A 1.a is interpreted as the three tokens 1, and a, while 1. a is interpreted as the two tokens 1. and a

## Character Set

Character:

any Unicode character

## End Of File

EndOfFile: physical end of the file \u0000 \u001A

The source text is terminated by whichever comes ﬁrst.

## End of Line

EndOfLine:

\u000D

\u000A

\u000D

\u000A

\u2028

\u2029

EndOfFile

There is no backslash line splicing, nor are there any limits on the length of a line.

## White Space

WhiteSpace:

Space

Space WhiteSpace

Space:

\u0020

\u0009

\u000B

\u000C

## Comment Block

A Comment Block group is starts with a /\* and ends when a \*/ is read. It has the following format:

|  |  |  |
| --- | --- | --- |
| /\* | ... | \*/ |

When the system is reading a Comment Block, text will be analyzed as a series of characters. So, anytime a \*/ is read, the group will end. This includes cases where the \*/ is embedded in other lexical structures such as string literals.

No other groups can be nested within this one.

## Comment Line

A Comment Line group is starts with a // and ends when a NewLine is read. It has the following format:

|  |  |  |
| --- | --- | --- |
| // | ... | NewLine |

When the system is reading a Comment Line, text will be analyzed as a series of characters. So, anytime a NewLine is read, the group will end. This includes cases where the NewLine is embedded in other lexical structures such as string literals. The NewLine is not considered part of the group.

No other groups can be nested within this one.

## Special Symbols

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

V# supports user-special symbols by giving a providing special literals.

V# has a total of 56 special symbols.

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

## Reserved Words

V# has a total of 98 reserved words.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **abstract** | **delegate** | **include** | **readonly** | **this** |
| **add** | **delete** | **int** | **ref** | **throw** |
| **adressof** | **do** | **interface** | **reg** | **try** |
| **as** | **double** | **interrupt** | **remove** | **type** |
| **assembly** | **else** | **is** | **restrict** | **typeof** |
| **assert** | **enum** | **long** | **return** | **uint** |
| **async** | **event** | **mixin** | **sbyte** | **ulong** |
| **bool** | **explicit** | **nameof** | **sealed** | **unchecked** |
| **break** | **extern** | **new** | **self** | **union** |
| **byte** | **finally** | **object** | **set** | **unsafe** |
| **case** | **float** | **operator** | **shared** | **ushort** |
| **catch** | **for** | **out** | **short** | **using** |
| **char** | **foreach** | **override** | **sizeof** | **virtual** |
| **checked** | **friend** | **package** | **static** | **vlong** |
| **class** | **get** | **params** | **string** | **void** |
| **complex** | **goto** | **pass** | **struct** | **where** |
| **const** | **if** | **private** | **super** | **while** |
| **continue** | **implicit** | **protected** | **supersede** | **yield** |
| **default** | **import** | **public** | **switch** |  |
| **define** | **in** | **raise** | **sync** |  |

## iDENTIFIERS

An identifier is an unlimited-length sequence of letters and digits, the first of which must be a letter.

It must start with a letter followed by a combination of alpha-numeric characters (‘\_’ included).

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

An identifier cannot have the same spelling (Unicode character sequence) as a keyword, Boolean literal, or the null literal, or a compile-time error occurs.

Two identifiers are the same only if they are identical, that is, have the same Unicode character for each letter or digit. Identifiers that have the same external appearance may yet be different.

## LITERALS

A literal is the source code representation of a value of a primitive type, the String type, or the null type.

A string character is sequence of printable characters except “.

A character is a printable character except ‘.

A hexadecimal digit is a [0..9] or [A..F] or [a..f] character

An octal digit is a [0..7] character.

A binary digit is 0 or 1.

## integral literal

An integer literal may be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2).

In a hexadecimal or binary literal, the integer is only denoted by the digits after the 0x or 0b characters and before any type suffix.

In a decimal or octal literal, the integer is denoted by all the digits in the literal before any type suffix.

|  |  |
| --- | --- |
| [***Integral-Literal***](#integral_literal) **:** | |
|  | **BinaryLiteral** |
|  | **OctLiteral** |
|  | **DecLiteral** |
|  | **HexLiteral** |
|  |  |

The type of an integer literal is determined as follows:

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

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

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

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

* When a decimal-integer-literal with the value 2147483648 (231) and no integer-type-suffix appears as the token immediately following a unary minus operator token, the result is a constant of type int with the value −2147483648 (−231). In all other situations, such a decimal-integer-literal is of type uint.
* When a decimal-integer-literal with the value 9223372036854775808 (263) and no integer-type-suffix or the integer-type-suffix L or l appears as the token immediately following a unary minus operator token, the result is a constant of type long with the value −9223372036854775808 (−263). In all other situations, such a decimal-integer-literal is of type ulong.

### Decimal Literal

A decimal numeral is either the single ASCII digit 0, representing the integer zero, or consists of an ASCII digit from 1 to 9 optionally followed by one or more ASCII digits from 0 to 9 interspersed with underscores, representing a positive integer.

INTEGER\_LITERAL: [0-9]+ IntegerTypeSuffix?;

IntegerTypeSuffix: [lL]? [uU] | [uU]? [lL];

### Octal Literal

An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7 and can represent a positive, zero, or negative integer.

OCT\_INTEGER\_LITERAL: '0' [oO] OctDigit+ IntegerTypeSuffix?;

OctDigit : [0-7];

### Hexadecimal Literal

A hexadecimal numeral consists of the leading ASCII characters 0x or 0X followed by one or more ASCII hexadecimal digits and can represent a positive, zero, or negative integer.

Hexadecimal digits with values 10 through 15 are represented by the ASCII letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

HEX\_INTEGER\_LITERAL: '0' [xX] HexDigit+ IntegerTypeSuffix?;

HexDigit : [0-9] | [A-F] | [a-f];

### Binary Literal

A binary numeral consists of the leading ASCII characters 0b or 0B followed by one or more of the ASCII digits 0 or 1 and can represent a positive, zero, or negative integer.

BIN\_INTEGER\_LITERAL: '0' [bB] BinDigit+ IntegerTypeSuffix?;

BinDigit : [0-1];

## Real Literal

A floating-point literal has the following parts: a whole-number part, a decimal or hexadecimal point (represented by an ASCII period character), a fraction part, an exponent, and a type suffix.

A floating-point literal may be expressed in decimal (base 10) or hexadecimal (base 16).

For decimal floating-point literals, at least one digit (in either the whole number or the fraction part) and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer.

A floating-point literal is of type float if it is suffixed with an ASCII letter F or f, if its type is double and it can optionally be suffixed with an ASCII letter D or d, otherwise its type is double imaginary and it can optionally be suffixed with an ASCII letter I or i.

The elements of the types float and double are those values that can be represented using the IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point formats, respectively.

The largest positive finite literal of type float is 3.4028235e38f.

The smallest positive finite non-zero literal of type float is 1.40e-45f.

The largest positive finite literal of type double is 1.7976931348623157e308.

The smallest positive finite non-zero literal of type double is 4.9e-324.

It is a compile-time error if a non-zero floating-point literal is too large, so that on rounded conversion to its internal representation, it becomes an IEEE 754 infinity.

A program can represent infinities without producing a compile-time error by using constant expressions such as 1f/0f or -1d/0d or by using the predefined constants POSITIVE\_INFINITY and NEGATIVE\_INFINITY of the classes Float and Double.

It is a compile-time error if a non-zero floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero.

A compile-time error does not occur if a non-zero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a nonzero deformalized number.

REAL\_LITERAL:

[0-9]\* '.' [0-9]+ ExponentPart? [FfDdMm]?

| [0-9]+ ([FfDdMm] | ExponentPart [FfDdMm]?);

ExponentPart: [eE] ('+' | '-')? [0-9]+;

## Boolean Literal

The boolean type has two values, represented by the boolean literals true and false, formed from ASCII letters.

BOOL\_LITERAL : true | false;

A boolean literal is always of type boolean.

## Null Literal

The null type has one value, the null reference, represented by the null literal null, which is formed from ASCII characters.

NULL\_LITERAL: null;

A null literal is always of the null type.

## CHARACTER LITERAL

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes. (The single-quote, or apostrophe, character is \u0027.)

CHARACTER\_LITERAL:'\'' (~['\\\r\n\u0085\u2028\u2029] | CommonCharacter) '\'';

CommonCharacter

: SimpleEscapeSequence

| HexEscapeSequence

| UnicodeEscapeSequence

;

SimpleEscapeSequence

: '\\\''

| '\\"'

| '\\\\'

| '\\0'

| '\\a'

| '\\b'

| '\\f'

| '\\n'

| '\\r'

| '\\t'

| '\\v'

;

HexEscapeSequence

: '\\x' HexDigit

| '\\x' HexDigit HexDigit

| '\\x' HexDigit HexDigit HexDigit

| '\\x' HexDigit HexDigit HexDigit HexDigit

;

UnicodeEscapeSequence

: '\\u' HexDigit HexDigit HexDigit HexDigit

| '\\U' HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit

;

HexDigit : [0-9] | [A-F] | [a-f];

Character literals can only represent UTF-16 code units, i.e., they are limited to values from \u0000 to \uffff. Supplementary characters must be represented.

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

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

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

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

The type of a character-literal is char.

## String Literal

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

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

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

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

The example

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

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

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

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

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

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

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.

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

class Test  
{

[EntryPoint]  
 static void Main() {  
 object a = "hello";  
 object b = "hello";  
 WriteLn(a == b);  
 }  
}

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

REGULAR\_STRING: '"' (~["\\\r\n\u0085\u2028\u2029] | CommonCharacter)\* '"';

VERBATIUM\_STRING: '@"' (~'"' | '""')\* '"';

## Operators literal

V# provides a way to define custom operators like ‘+’, ‘-‘… by using literals. This concept allow developer to simplify expressions by replacing method calls with operators.

For example :

Matrix a,b;

b = (1 / det(a)) \* transpose(com(a));

can be replace by:

b = (1 / @$a) \* @+(@-a) ;

where:

@$ stands for det

@+ stands for transpose

@- stands for com

These operators aren’t included by default but when they are overridden the compiler will consider them as valid operators.

The use of undefined (not overridden) operator will cause a compile-time error.

### Binary Operator Literal

A binary operator literal is a sequence of special symbols that starts always with ‘@@’.

BINARY\_OPERATOR\_LITERAL: '@@' OpSymbol+;

OpSymbol : '+' | '-' | '\*' | '/' | '%' | '<' | '>' | '~' | '^' | '&' | '!' | '|' | '='

### Unary Operator Literal

A unary operator literal is a sequence of special symbols that starts always with ‘@’.

UNARY\_OPERATOR\_LITERAL: '@' OpSymbol+;

## Sepearators

Twelve tokens, formed from ASCII characters, are the separators (punctuators).

Separator:

(one of) ( ) { } [ ] ; , . ... @ ::

3

# GRAMMAR

**T**his specification presents the syntax of the V# programming language using two grammars. The **lexical grammar** defines how Unicode characters are combined to form line terminators, white space, comments and tokens. The **syntactic grammar** defines how the tokens resulting from the lexical grammar are combined to form V# programs.

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 a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols.

## SYNTAX

// entry point

compilation\_unit

: BYTE\_ORDER\_MARK? import\_directives?

global\_attribute\_section\* package\_member\_declarations? EOF

;

//B.2 Syntactic grammar

//B.2.1 Basic concepts

package\_or\_type\_name

: (identifier type\_argument\_list? | qualified\_alias\_member) ('.' identifier type\_argument\_list?)\*

;

//B.2.2 Types

type

: base\_type ('?' | rank\_specifier | '\*')\*

;

base\_type

: simple\_type

| class\_type // represents types: enum, class, interface, delegate, type\_parameter

| VOID '\*'

;

simple\_type

: numeric\_type

| BOOL

;

*numeric\_type*

*: integral\_type*

*| floating\_point\_type*

;

integral\_type

: SBYTE

| BYTE

| SHORT

| USHORT

| INT

| UINT

| LONG

| ULONG

| VLONG

| CHAR

;

floating\_point\_type

: FLOAT

| DOUBLE

| COMPLEX

;

class\_type

: package\_or\_type\_name

| OBJECT

| STRING

;

type\_argument\_list

: '<' type ( ',' type)\* '>'

;

//B.2.4 Expressions

argument\_list

: argument ( ',' argument)\*

;

argument

: (identifier ':')? refout=(REF | OUT)? expression

;

expression

: assignment

| non\_assignment\_expression

;

non\_assignment\_expression

: lambda\_expression

| conditional\_expression

;

assignment

: unary\_expression assignment\_operator expression

;

assignment\_operator

: '=' | '+=' | '-=' | '\*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<=' | '<~=' | '~>=' | right*\_shift\_assignment*

*;*

conditional\_expression

: null\_coalescing\_expression ('?' expression ':' expression)?

;

null\_coalescing\_expression

: binary\_operator\_expression ('??' null\_coalescing\_expression)?

;

binary\_operator\_expression

: conditional\_or\_expression (BINARY\_OPERATOR\_LITERAL conditional\_or\_expression)\*

;

conditional\_or\_expression

: conditional\_and\_expression (OP\_OR conditional\_and\_expression)\*

;

conditional\_and\_expression

: inclusive\_or\_expression (OP\_AND inclusive\_or\_expression)\*

;

inclusive\_or\_expression

: exclusive\_or\_expression ('|' exclusive\_or\_expression)\*

;

exclusive\_or\_expression

: and\_expression ('^' and\_expression)\*

;

and\_expression

: equality\_expression ('&' equality\_expression)\*

;

equality\_expression

: relational\_expression ((OP\_EQ | OP\_NE) relational\_expression)\*

;

relational\_expression

: shift\_expression (('<' | '>' | '<=' | '>=') shift\_expression | IS isType | AS type)\*

;

shift\_expression

: additive\_expression (('<<' | right\_shift | '<~' | '~>' ) additive\_expression)\*

;

additive\_expression

: multiplicative\_expression (('+' | '-') multiplicative\_expression)\*

;

multiplicative\_expression

: unary\_expression (('\*' | '/' | '%') unary\_expression)\*

;

unary\_expression

: primary\_expression

| '+' unary\_expression

| '-' unary\_expression

| BANG unary\_expression

| '~' unary\_expression

| '++' unary\_expression

| '--' unary\_expression

| '?!' unary\_expression

| '?~' unary\_expression

| OPEN\_PARENS type CLOSE\_PARENS unary\_expression

| '&' unary\_expression

| '\*' unary\_expression

| UNARY\_OPERATOR\_LITERAL unary\_expression

;

primary\_expression // Null-conditional operators

: pe=primary\_expression\_start bracket\_expression\*

((member\_access | method\_invocation | '++' | '--' | '->' identifier) bracket\_expression\*)\*

;

primary\_expression\_start

: literal

| identifier type\_argument\_list?

| OPEN\_PARENS expression CLOSE\_PARENS

| predefined\_type

| qualified\_alias\_member

| LITERAL\_ACCESS

| SELF

| SUPER ('.' identifier type\_argument\_list? | '[' expression\_list ']')

| NEW (type (object\_creation\_expression

| object\_or\_collection\_initializer

| '[' expression\_list ']' rank\_specifier\* array\_initializer?

| rank\_specifier+ array\_initializer)

| anonymous\_object\_initializer

| rank\_specifier array\_initializer)

| TYPEOF OPEN\_PARENS (unbound\_type\_name | type | VOID) CLOSE\_PARENS

| CHECKED OPEN\_PARENS expression CLOSE\_PARENS

| UNCHECKED OPEN\_PARENS expression CLOSE\_PARENS

| DEFAULT OPEN\_PARENS type CLOSE\_PARENS

| DELEGATE (OPEN\_PARENS explicit\_anonymous\_function\_parameter\_list? CLOSE\_PARENS)? block

| SIZEOF OPEN\_PARENS type CLOSE\_PARENS #sizeofExpression

| ADRESSOF OPEN\_PARENS expression CLOSE\_PARENS

| NAMEOF OPEN\_PARENS (identifier '.')\* identifier CLOSE\_PARENS

;

member\_access

: '?'? '.' identifier type\_argument\_list?

;

bracket\_expression

: '?'? '[' indexer\_argument ( ',' indexer\_argument)\* ']'

;

indexer\_argument

: (identifier ':')? expression

;

predefined\_type

: BOOL | BYTE | CHAR | DOUBLE | FLOAT | INT | LONG

| OBJECT | SBYTE | SHORT | STRING | UINT | ULONG | VLONG | USHORT

;

expression\_list

: expression (',' expression)\*

;

object\_or\_collection\_initializer

: object\_initializer

| collection\_initializer

;

object\_initializer

: OPEN\_BRACE (member\_initializer\_list ','?)? CLOSE\_BRACE

;

member\_initializer\_list

: member\_initializer (',' member\_initializer)\*

;

member\_initializer

: (identifier | '[' expression ']') '=' initializer\_value

;

initializer\_value

: expression

| object\_or\_collection\_initializer

;

collection\_initializer

: OPEN\_BRACE element\_initializer (',' element\_initializer)\* ','? CLOSE\_BRACE

;

element\_initializer

: non\_assignment\_expression

| OPEN\_BRACE expression\_list CLOSE\_BRACE

;

anonymous\_object\_initializer

: OPEN\_BRACE (member\_declarator\_list ','?)? CLOSE\_BRACE

;

member\_declarator\_list

: member\_declarator ( ',' member\_declarator)\*

;

member\_declarator

: primary\_expression

| identifier '=' expression

;

unbound\_type\_name

: identifier ( generic\_dimension\_specifier? | '::' identifier generic\_dimension\_specifier?)

('.' identifier generic\_dimension\_specifier?)\*

;

generic\_dimension\_specifier

: '<' ','\* '>'

;

isType

: base\_type (rank\_specifier | '\*')\* '?'?

;

lambda\_expression

: anonymous\_function\_signature right\_arrow anonymous\_function\_body

;

anonymous\_function\_signature

: OPEN\_PARENS CLOSE\_PARENS

| OPEN\_PARENS explicit\_anonymous\_function\_parameter\_list CLOSE\_PARENS

| OPEN\_PARENS implicit\_anonymous\_function\_parameter\_list CLOSE\_PARENS

| identifier

;

explicit\_anonymous\_function\_parameter\_list

: explicit\_anonymous\_function\_parameter ( ',' explicit\_anonymous\_function\_parameter)\*

;

explicit\_anonymous\_function\_parameter

: refout=(REF | OUT)? type identifier

;

implicit\_anonymous\_function\_parameter\_list

: identifier (',' identifier)\*

;

anonymous\_function\_body

: expression

| block

;

//B.2.5 Statements

statement

: identifier ':' statement #labeledStatement

| (local\_variable\_declaration | local\_constant\_declaration) ';' #declarationStatement

| embedded\_statement #embeddedStatement

;

embedded\_statement

: block

| simple\_embedded\_statement

;

simple\_embedded\_statement

: ';'

| expression ';'

// selection statements

| IF OPEN\_PARENS expression CLOSE\_PARENS if\_body (ELSE if\_body)?

| SWITCH OPEN\_PARENS expression CLOSE\_PARENS OPEN\_BRACE switch\_section\* CLOSE\_BRACE

// iteration statements

| WHILE OPEN\_PARENS expression CLOSE\_PARENS embedded\_statement

| WHILE OPEN\_PARENS expression CLOSE\_PARENS embedded\_statement ELSE embedded\_statement

| DO embedded\_statement WHILE OPEN\_PARENS expression CLOSE\_PARENS ';'

| FOR OPEN\_PARENS for\_initializer? ';' expression? ';' for\_iterator? CLOSE\_PARENS embedded\_statement

| FOREACH OPEN\_PARENS type identifier IN expression CLOSE\_PARENS embedded\_statement

// jump statements

| BREAK ';'

| CONTINUE ';'

| GOTO (identifier | CASE expression | DEFAULT) ';'

| RETURN expression? ';'

| THROW expression? ';'

| TRY block (catch\_clauses finally\_clause? | finally\_clause)

| CHECKED block

| UNCHECKED block

| SYNC OPEN\_PARENS expression CLOSE\_PARENS embedded\_statement

| USING OPEN\_PARENS resource\_acquisition CLOSE\_PARENS embedded

| YIELD (RETURN expression | BREAK) ';'

| MIXIN identifier ';'

| RESTRICT OPEN\_PARENS expression CLOSE\_PARENS embedded\_statement

// unsafe statements

| UNSAFE block

;

block

: OPEN\_BRACE statement\_list? CLOSE\_BRACE

;

local\_variable\_declaration

: type local\_variable\_declarator ( ',' local\_variable\_declarator)\*

;

local\_variable\_declarator

: identifier ('=' local\_variable\_initializer)?

;

local\_variable\_initializer

: expression

| array\_initializer

;

local\_constant\_declaration

: CONST type constant\_declarators

;

if\_body

: block

| simple\_embedded\_statement

;

switch\_section

: switch\_label+ statement\_list

;

switch\_label

: CASE expression ':'

| DEFAULT ':'

;

statement\_list

: statement+

;

for\_initializer

: local\_variable\_declaration

| expression (',' expression)\*

;

for\_iterator

: expression (',' expression)\*

;

catch\_clauses

: specific\_catch\_clause (specific\_catch\_clause)\* general\_catch\_clause?

| general\_catch\_clause

;

specific\_catch\_clause

: CATCH OPEN\_PARENS class\_type identifier? CLOSE\_PARENS exception\_filter? block

;

general\_catch\_clause

: CATCH exception\_filter? block

;

exception\_filter

: WHEN OPEN\_PARENS expression CLOSE\_PARENS

;

finally\_clause

: FINALLY block

;

resource\_acquisition

: local\_variable\_declaration

| expression

;

//B.2.6 Packages;

package\_declaration

: PACKAGE qi=qualified\_identifier package\_body ';'?

;

qualified\_identifier

: identifier ( '.' identifier )\*

;

package\_body

: OPEN\_BRACE import\_directives? package\_member\_declarations? CLOSE\_BRACE

;

import\_directives

: import\_directive+

;

import\_directive

: IMPORT identifier '=' package\_or\_type\_name ';'

| IMPORT package\_or\_type\_name ';'

| IMPORT STATIC package\_or\_type\_name ';'

;

package\_member\_declarations

: package\_member\_declaration+

;

package\_member\_declaration

: package\_declaration

| type\_declaration

| attributes? all\_member\_modifiers? (default\_member\_declaration)

;

default\_member\_declaration

: constant\_declaration

| typed\_member\_declaration

| event\_declaration

| conversion\_operator\_declarator (body | right\_arrow expression ';')

| VOID method\_declaration

| mixin\_definition

| interrupt\_definition

;

type\_declaration

: attributes? all\_member\_modifiers?

(class\_definition | struct\_definition | union\_definition | interface\_definition | enum\_definition | delegate\_definition)

;

qualified\_alias\_member

: identifier '::' identifier type\_argument\_list?

;

//B.2.7 Classes;

type\_parameter\_list

: '<' type\_parameter (',' type\_parameter)\* '>'

;

type\_parameter

: attributes? identifier

;

class\_base

: ':' class\_type (',' package\_or\_type\_name)\*

;

interface\_type\_list

: package\_or\_type\_name (',' package\_or\_type\_name)\*

;

type\_parameter\_constraints\_clauses

: type\_parameter\_constraints\_clause+

;

type\_parameter\_constraints\_clause

: WHERE identifier ':' type\_parameter\_constraints

;

type\_parameter\_constraints

: primary\_constraint (',' secondary\_constraints)?

;

primary\_constraint

: class\_type

| CLASS

| STRUCT

;

secondary\_constraints

: package\_or\_type\_name (',' package\_or\_type\_name)\*

;

class\_body

: OPEN\_BRACE class\_member\_declarations? CLOSE\_BRACE

;

class\_member\_declarations

: class\_member\_declaration+

;

class\_member\_declaration

: attributes? all\_member\_modifiers? (common\_member\_declaration | destructor\_definition)

;

all\_member\_modifiers

: all\_member\_modifier+

;

all\_member\_modifier

: NEW

| PUBLIC

| PROTECTED

| FRIEND

| PRIVATE

| READONLY

| VOLATILE

| VIRTUAL

| SEALED

| OVERRIDE

| ABSTRACT

| STATIC

| UNSAFE

| EXTERN

| PARTIAL

| SUPERSEDE

| INLINE

| SYNC

;

common\_member\_declaration

: constant\_declaration

| typed\_member\_declaration

| event\_declaration

| conversion\_operator\_declarator (body | right\_arrow expression ';')

| constructor\_declaration

| VOID method\_declaration

| class\_definition

| struct\_definition

| interface\_definition

| enum\_definition

| mixin\_definition

| delegate\_definition

;

typed\_member\_declaration

: type

( package\_or\_type\_name '.' indexer\_declaration

| method\_declaration

| property\_declaration

| indexer\_declaration

| operator\_declaration

| field\_declaration

)

;

constant\_declarators

: constant\_declarator (',' constant\_declarator)\*

;

constant\_declarator

: identifier '=' expression

;

variable\_declarators

: variable\_declarator (',' variable\_declarator)\*

;

variable\_declarator

: identifier ('=' variable\_initializer)?

;

variable\_initializer

: expression

| array\_initializer

;

return\_type

: type

| VOID

;

member\_name

: package\_or\_type\_name

;

method\_body

: block

| ';'

;

formal\_parameter\_list

: parameter\_array

| fixed\_parameters (',' parameter\_array)?

;

fixed\_parameters

: fixed\_parameter ( ',' fixed\_parameter )\*

;

fixed\_parameter

: attributes? parameter\_modifier? arg\_declaration

| ARGLIST

;

parameter\_modifier

: REF

| OUT

| SELF

| YIELD

;

parameter\_array

: attributes? PARAMS array\_type identifier

;

accessor\_declarations

: attrs=attributes? mods=accessor\_modifier?

(GET accessor\_body set\_accessor\_declaration? | SET accessor\_body get\_accessor\_declaration?)

;

get\_accessor\_declaration

: attributes? accessor\_modifier? GET accessor\_body

;

set\_accessor\_declaration

: attributes? accessor\_modifier? SET accessor\_body

;

accessor\_modifier

: PROTECTED

| FRIEND

| PRIVATE

| PROTECTED FRIEND

| FRIEND PROTECTED

;

accessor\_body

: block

| ';'

;

event\_accessor\_declarations

: attributes? (ADD block remove\_accessor\_declaration | REMOVE block add\_accessor\_declaration) raise\_accessor\_declaration?

;

add\_accessor\_declaration

: attributes? ADD block

;

remove\_accessor\_declaration

: attributes? REMOVE block

;

raise\_accessor\_declaration

: attributes? RAISE block

;

overloadable\_operator

: '+'

| '-'

| BANG

| '~'

| '++'

| '--'

| TRUE

| FALSE

| '\*'

| '/'

| '%'

| '&'

| '|'

| '^'

| '<<'

| right\_shift

| OP\_EQ

| OP\_NE

| '>'

| '<'

| '>='

| '<='

| BINARY\_OPERATOR\_LITERAL

| UNARY\_OPERATOR\_LITERAL

;

conversion\_operator\_declarator

: (IMPLICIT | EXPLICIT) OPERATOR type OPEN\_PARENS arg\_declaration CLOSE\_PARENS

;

constructor\_initializer

: ':' (SUPER | SELF) OPEN\_PARENS argument\_list? CLOSE\_PARENS

;

body

: block

| ';'

;

//B.2.8 Structs

struct\_interfaces

: ':' interface\_type\_list

;

struct\_body

: OPEN\_BRACE struct\_member\_declaration\* CLOSE\_BRACE

;

struct\_member\_declaration

: attributes? all\_member\_modifiers?

( common\_member\_declaration )

;

//B.2.9 Arrays

array\_type

: base\_type (('\*' | '?')\* rank\_specifier)+

;

rank\_specifier

: '[' ','\* ']'

;

array\_initializer

: OPEN\_BRACE (variable\_initializer (',' variable\_initializer)\* ','?)? CLOSE\_BRACE

;

//B.2.10 Interfaces

variant\_type\_parameter\_list

: '<' variant\_type\_parameter (',' variant\_type\_parameter)\* '>'

;

variant\_type\_parameter

: attributes? variance\_annotation? identifier

;

variance\_annotation

: IN | OUT

;

interface\_base

: ':' interface\_type\_list

;

interface\_body

: OPEN\_BRACE interface\_member\_declaration\* CLOSE\_BRACE

;

interface\_member\_declaration

: attributes? NEW?

(UNSAFE? type

( identifier type\_parameter\_list? OPEN\_PARENS formal\_parameter\_list? CLOSE\_PARENS type\_parameter\_constraints\_clauses? ';'

| identifier OPEN\_BRACE interface\_accessors CLOSE\_BRACE

| SELF '[' formal\_parameter\_list ']' OPEN\_BRACE interface\_accessors CLOSE\_BRACE)

| UNSAFE? VOID identifier type\_parameter\_list? OPEN\_PARENS formal\_parameter\_list? CLOSE\_PARENS type\_parameter\_constraints\_clauses? ';'

| EVENT type identifier ';')

;

interface\_accessors

: attributes? (GET ';' (attributes? SET ';')? | SET ';' (attributes? GET ';')?)

;

//B.2.11 Enums

enum\_base

: ':' integral\_type

;

enum\_body

: OPEN\_BRACE (enum\_member\_declaration (',' enum\_member\_declaration)\* ','?)? CLOSE\_BRACE

;

enum\_member\_declaration

: attributes? identifier ('=' expression)?

;

//B.2.12 Delegates

//B.2.13 Attributes

global\_attribute\_section

: '[' global\_attribute\_target ':' attribute\_list ','? ']'

;

global\_attribute\_target

: keyword

| identifier

;

attributes

: attribute\_section+

;

attribute\_section

: '[' (attribute\_target ':')? attribute\_list ','? ']'

;

attribute\_target

: keyword

| identifier

;

attribute\_list

: attribute (',' attribute)\*

;

attribute

: package\_or\_type\_name (OPEN\_PARENS (attribute\_argument (',' attribute\_argument)\*)? CLOSE\_PARENS)?

;

attribute\_argument

: (identifier ':')? expression

;

//B.3 Grammar extensions for unsafe code

pointer\_type

: (simple\_type | class\_type) (rank\_specifier | '?')\* '\*'

| VOID '\*'

;

right\_arrow

: first='=' second='>' {$first.index + 1 == $second.index}? // Nothing between the tokens?

;

right\_shift

: first='>' second='>' {$first.index + 1 == $second.index}? // Nothing between the tokens?

;

right\_shift\_assignment

: first='>' second='>=' {$first.index + 1 == $second.index}? // Nothing between the tokens?

;

literal

: boolean\_literal

| string\_literal

| INTEGER\_LITERAL

| HEX\_INTEGER\_LITERAL

| OCT\_INTEGER\_LITERAL

| BIN\_INTEGER\_LITERAL

| REAL\_LITERAL

| CHARACTER\_LITERAL

| NULL

;

boolean\_literal

: TRUE

| FALSE

;

string\_literal

: interpolated\_regular\_string

| interpolated\_verbatium\_string

| REGULAR\_STRING

| VERBATIUM\_STRING

;

interpolated\_regular\_string

: INTERPOLATED\_REGULAR\_STRING\_START interpolated\_regular\_string\_part\* DOUBLE\_QUOTE\_INSIDE

;

interpolated\_verbatium\_string

: INTERPOLATED\_VERBATIUM\_STRING\_START interpolated\_verbatium\_string\_part\* DOUBLE\_QUOTE\_INSIDE

;

interpolated\_regular\_string\_part

: interpolated\_string\_expression

| DOUBLE\_CURLY\_INSIDE

| REGULAR\_CHAR\_INSIDE

| REGULAR\_STRING\_INSIDE

;

interpolated\_verbatium\_string\_part

: interpolated\_string\_expression

| DOUBLE\_CURLY\_INSIDE

| VERBATIUM\_DOUBLE\_QUOTE\_INSIDE

| VERBATIUM\_INSIDE\_STRING

;

interpolated\_string\_expression

: expression (',' expression)\* (':' FORMAT\_STRING+)?

;

//B.1.7 Keywords

keyword

: ABSTRACT

| AS

| SUPER

| BOOL

| BREAK

| BYTE

| CASE

| CATCH

| CHAR

| CHECKED

| CLASS

| CONST

| CONTINUE

| COMPLEX

| DEFAULT

| DELEGATE

| DO

| DOUBLE

| ELSE

| ENUM

| EVENT

| EXPLICIT

| EXTERN

| FALSE

| FINALLY

| FLOAT

| FOR

| FOREACH

| GOTO

| IF

| IMPLICIT

| IN

| INT

| INTERFACE

| FRIEND

| IS

| SYNC

| LONG

| PACKAGE

| NEW

| NULL

| MIXIN

| OBJECT

| OPERATOR

| OUT

| OVERRIDE

| PARAMS

| PRIVATE

| PROTECTED

| PUBLIC

| READONLY

| REF

| RETURN

| SBYTE

| SEALED

| SHORT

| SIZEOF

| STATIC

| STRING

| STRUCT

| SWITCH

| SELF

| THROW

| TRUE

| TRY

| TYPEOF

| UINT

| ULONG

| UNCHECKED

| UNSAFE

| UNION

| USHORT

| IMPORT

| USING

| VIRTUAL

| VOID

| VOLATILE

| WHILE

;

class\_definition

: CLASS identifier type\_parameter\_list? class\_base? type\_parameter\_constraints\_clauses?

class\_body ';'?

;

mixin\_definition

: MIXIN identifier block

;

interrupt\_definition

: INTERRUPT (INTEGER\_LITERAL | HEX\_INTEGER\_LITERAL) block

;

struct\_definition

: STRUCT identifier type\_parameter\_list? struct\_interfaces? type\_parameter\_constraints\_clauses?

struct\_body ';'?

;

union\_definition

: UNION identifier type\_parameter\_list? struct\_interfaces? type\_parameter\_constraints\_clauses?

struct\_body ';'?

;

interface\_definition

: INTERFACE identifier variant\_type\_parameter\_list? interface\_base?

type\_parameter\_constraints\_clauses? interface\_body ';'?

;

enum\_definition

: ENUM identifier enum\_base? enum\_body ';'?

;

delegate\_definition

: DELEGATE return\_type identifier variant\_type\_parameter\_list?

OPEN\_PARENS formal\_parameter\_list? CLOSE\_PARENS type\_parameter\_constraints\_clauses? ';'

;

event\_declaration

: EVENT type (variable\_declarators ';' | member\_name OPEN\_BRACE event\_accessor\_declarations CLOSE\_BRACE)

;

field\_declaration

: variable\_declarators ';'

;

property\_declaration // Property initializer & lambda in properties C# 6

: member\_name (OPEN\_BRACE accessor\_declarations CLOSE\_BRACE ('=' variable\_initializer ';')? | right\_arrow expression ';')

;

constant\_declaration

: CONST type constant\_declarators ';'

;

indexer\_declaration

: SELF '[' formal\_parameter\_list ']' (OPEN\_BRACE accessor\_declarations CLOSE\_BRACE | right\_arrow expression ';')

;

destructor\_definition

: '~' SELF OPEN\_PARENS CLOSE\_PARENS body

;

constructor\_declaration

: SELF OPEN\_PARENS formal\_parameter\_list? CLOSE\_PARENS constructor\_initializer? body

;

method\_declaration

: method\_member\_name type\_parameter\_list? OPEN\_PARENS formal\_parameter\_list? CLOSE\_PARENS

type\_parameter\_constraints\_clauses? (method\_body | right\_arrow expression ';')

;

method\_member\_name

: (identifier | identifier '::' identifier) (type\_argument\_list? '.' identifier)\*

;

operator\_declaration

: OPERATOR overloadable\_operator OPEN\_PARENS arg\_declaration

(',' arg\_declaration)? CLOSE\_PARENS (body | right\_arrow expression ';')

;

arg\_declaration

: type identifier ('=' expression)?

;

method\_invocation

: OPEN\_PARENS argument\_list? CLOSE\_PARENS

;

object\_creation\_expression

: OPEN\_PARENS argument\_list? CLOSE\_PARENS object\_or\_collection\_initializer?

;

identifier

: IDENTIFIER

| ADD

| ARGLIST

| SET

| REMOVE

| RAISE

| NAMEOF

| GET

| ADRESSOF

| PARTIAL

| WHERE

;

4

# lifecycle

Every program must have an entry point. An entry point is where control is transferred from the [operating system](https://en.wikipedia.org/wiki/Operating_system) to a [computer program](https://en.wikipedia.org/wiki/Computer_program), at which place the processor enters a program or a [code](https://en.wikipedia.org/wiki/Machine_code) fragment and [execution](https://en.wikipedia.org/wiki/Execution_(computing)) begins.

## Startup

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

[EntryPoint]

static void Main() {...}

[EntryPoint]

static void Main(string[] args) {...}

[EntryPoint]

static int Main() {...}

[EntryPoint]

static int Main(string[] args) {...}

As shown, the entry point may optionally return an int value. This return value is used in application termination.

An entry point must have the method attribute EntryPoint so that the compiler recognizes it as a valid entry point.

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be string[]. If the formal parameter is present, the execution environment creates and passes a string[] argument containing the command-line arguments that were specified when the application was started. The string[] argument is never null, but it may have a length of zero if no command-line arguments were specified.

Since V# supports method overloading, a classes, structs or default declarations may contain multiple definitions of some method, provided each has a different signature. However, within a single program, no classes, structs or default declarations may contain more than one method called Main whose definition qualifies it to be used as an application entry point. Other overloaded versions of Main are permitted, however, provided they have more than one parameter, or their only parameter is other than type string[].

An application can be made up of multiple classes, structs or default declarations. It is possible for more than one of these classes, structs or default declarations to contain a method called Main whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these Main methods as the entry point.

The application entry point method may not be in a generic class declaration.

The application entry point method may be declared outside a type or package declaration.

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

## TERMINATION

**Program 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 (}) which terminates that method, or executing a return statement that has no expression, results in a termination status code of 0.

Prior to an application’s termination, destructors for all of its objects that have not yet been garbage collected are called.

5

# TYPES

**T**he V# programming language is a statically typed language, which means that every variable and every expression has a type that is known at compile time. The V# programming language is also a strongly typed language, because types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong static typing helps detect errors at compile time. The types of the V# programming language are divided into three categories: **value types**, **reference types** and **pointer types**. Both value types and reference types may be **generic types**, which take one or more **type parameters**. Type parameters can designate both value types and reference types.

*type:  
value-type  
reference-type   
type-parameter*

The value types are the Boolean type and the numeric types. The numeric types are the integral types byte, short, int, long, sbyte, ushort, uint, ulong, complex, vlong and char, and the floating-point types float and double. The reference types are class types, interface types, and array types. There is also a special null type. An object is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects. All objects, including arrays, support the methods of class Object. String literals are represented by String objects.

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.

## Value types

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

*base-type:  
class-type  
simple-type*

*simple-type:  
numeric-type*bool

*numeric-type:  
integral-type  
floating-point-type*decimal

*integral-type:*sbyte  
byte  
short  
ushort  
int  
uint  
long  
ulong

vlong  
char

*floating-point-type:*float  
double

Complex

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 reference type. For every non-nullable value type there is a corresponding nullable value type denoting the same set of values plus the value null.

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

## DEFAULT VALUES

All value types implicitly have 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, vlong the default value is 0.
* For char, the default value is '\x0000'.
* For float, the default value is 0.0f.
* For double, the default value is 0.0d.
* For complex, the default value is (0.0d , 0.0i).
* For bool, the default value is false.

For an enum types E, the default value is 0, converted to the type E.

For a struct types, 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-reference-type the default value is also known as the null value of the nullable reference type.

## STRUCT TYPES

A struct type is a value type that can declare constants, fields, methods, properties, indexers, operators, instance constructors, static constructors, and nested types.

## SIMPLE TYPES

V# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words. The table below describes the simple types sizes in bytes and their predefined class types:

|  |  |  |
| --- | --- | --- |
| **Reserved word** | **Size** | **Aliased type** |
| sbyte | 1 byte (signed) | Std.SByte |
| byte | 1 byte | Std.Byte |
| short | 2 bytes (signed) | Std.Short |
| ushort | 2 bytes | Std.UShort |
| int | 4 bytes (signed) | Std.Int |
| uint | 4 bytes | Std.UInt |
| long | 8 bytes (signed) | Std.Long |
| ulong | 8 bytes | Std.ULong |
| char | 2 bytes | Std.Char |
| float | 4 bytes | Std.Float |
| double | 8 bytes | Std.Double |
| bool | 1 byte | Std.Boolean |
| complex | 16 bytes | Std.Complex |

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. 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. It is not possible to have constants of other struct types, but a similar effect is provided by static readonly fields.

Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator.

## INTEGRAL TYPES

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

* The sbyte type represents signed 8-bit integers with values between –128 and 127.
* The byte type represents unsigned 8-bit integers with values between 0 and 255.
* The short type represents signed 16-bit integers with values between –32768 and 32767.
* The ushort type represents unsigned 16-bit integers with values between 0 and 65535.
* The int type represents signed 32-bit integers with values between –2147483648 and 2147483647.
* The uint type represents unsigned 32-bit integers with values between 0 and 4294967295.
* The long type represents signed 64-bit integers with values between –9223372036854775808 and 9223372036854775807.
* The ulong type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
* The vlong type represents unsigned 8\*n-bit integers with values between 0 and 28\*n - 1 where n is the size of the vlong number in bytes.
* The char type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible values for the char type corresponds to the Unicode character set. Although char has the same representation as ushort, not all operations permitted on one type are permitted on the other.

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

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

The checked and unchecked operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions. In a checked context, an overflow produces a compile-time error or causes a 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.

## FLOATING POINT TYPES

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

* Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two.
* Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example, 1.0 / 0.0 yields positive infinity, and –1.0 / 0.0 yields negative infinity.
* The Not-a-Number value, often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.
* The finite set of non-zero values of the form s × m × 2e, where s is 1 or −1, and m and e are determined by the particular floating-point type: For float, 0 < m < 224 and −149 ≤ e ≤ 104, and for double, 0 < m < 253 and −1075 ≤ e ≤ 970. Denormalized floating-point numbers are considered valid non-zero values.

The float type can represent values ranging from approximately 1.5 × 10−45 to 3.4 × 1038 with a precision of 7 digits.

The double type can represent values ranging from approximately 5.0 × 10−324 to 1.7 × 10308 with a precision of 15-16 digits.

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

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

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

* If the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
* If the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
* If a floating-point operation is invalid, the result of the operation becomes NaN.
* If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

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

## BOOLEAN TYPES

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

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

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 V#, such conversions are accomplished by explicitly comparing an integral or floating-point value to zero, or by explicitly comparing an object reference to null.

## ENUM TYPES

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which must be byte, sbyte, short, ushort, int, uint, long or ulong. The set of values of the enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations.

## complex typeS

The complex type is a 128-bit data type suitable for mathematical and geometrical calculations. The complex type can represent values ranging from (5.0 × 10−324 , 5.0 × 10−324 i) to (1.7 × 10308, 1.7 × 10308i).

A complex type is a concatenation of two double values the first represents the real part while the second represents the imaginary part.

A complex number can be represented by the sum of a real number (x) and an imaginary part (y\*i):  
x + y \* i  
The imaginary part (y\*i) is a factor of i, known as the imaginary unit, and which satisfies that:  
i2 = -1 .

## REFERENCE NULLABLE TYPES

A nullable reference type can represent all values of its **underlying type** plus an additional null value. A nullable reference type is written T&?, where T is the underlying type.

The underlying type of a nullable reference type cannot be a nullable reference type or a reference type. For example, int&?&? and string&? are invalid types.

## reference types

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

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.

## Class types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize super classes. Instances of class types are created using object-creation-expressions. Certain predefined class types have special meaning in the V# language, as described in the table below.

|  |  |
| --- | --- |
| **Class type** | **Description** |
| Std.Object | The super class of all reference types. |
| Std.String | The string type of the V# language. |
| Std.Array | The super class of all array types. |
| Std.Delegate | The super class of all delegate types. |
| Std.Exception | The super class of all exception types. |

## OBJECT TYPES

The object class type is the ultimate super class of all other types. Every reference type in V# directly or indirectly derives from the object class type. The keyword object is simply an alias for the predefined class Std.Object.

## STRING TYPES

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

## INTERFACE TYPES

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

## 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.

## DELEGATE TYPES

A delegate is a data structure that refers to one or more methods. For instance methods, it also refers to their corresponding object instances. The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can 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.

## pointer types

Pointer types do not inherit from [object](https://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx) and no conversions exist between pointer types and object. Also, boxing and unboxing do not support pointers. However, you can convert between different pointer types and between pointer types and integral types.

A pointer cannot point to a reference or to a [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx) that contains references, because an object reference can be garbage collected even if a pointer is pointing to it. The garbage collector does not keep track of whether an object is being pointed to by any pointer types.

The value of the pointer variable of type myType\* is the address of a variable of type myType.

## Boxing and unboxing

The concept of boxing and unboxing is central to V#’s type system. It provides 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

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 enum-type to the type System.Enum.

Note that an implicit conversion from a type parameter will be executed as a boxing conversion if at run-time it ends up converting from a value type to a reference type.

## UNBOXING

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.Enum to any enum-type.

Note that an explicit conversion to a type parameter will be executed as an unboxing conversion if at run-time it ends up converting from a reference type to a value type.

## Constructed types

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

Constructed types can also be used in expressions as simple names or when accessing a member.

When a Non-array-type 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.

## TYPE ARGUMENTS

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

*type-generics-Decl:*<( *type- arguments)*>

*type-arguments:  
type   
type-arguments* , *type*

## 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.

## 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.

## 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. For each where clause, the type argument A that corresponds to the named type parameter is checked against each constraint as follows:

If the constraint is a class type, an interface type, or a type parameter, let C represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type A is convertible to type C by one of the following:

* An identity conversion.
* An implicit reference conversion.

If the constraint is the reference type constraint (class), the type A must satisfy one of the following:

* A is an interface type, class type, delegate type or array type.
* A is a type parameter that is known to be a reference type.

If the constraint is the value type constraint (struct), the type A must satisfy one of the following:

* A is a struct type or enum type, but not a nullable reference type.
* A is a type parameter having the value type constraint.

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

## 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 actual type arguments, type parameters have slightly different operations and restrictions than other types. These include:

* A type parameter cannot be used directly to declare a super class or interface.
* The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter.
* The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter.
* 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. However, a default expression can be used instead. In addition, a value with a type given by a type parameter can be compared with null using == and != unless the type parameter has the value type constraint.
* A type parameter cannot be used anywhere within an attribute.
* A type parameter cannot be used in a member access or type name to identify a static member.

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. The run-time execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

6

# VARIABLES

***V***ariables represent storage locations. Every variable has a type that determines what values can be stored in the variable. V# is a type-safe language, and the V# 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 operators.

A variable must be definitely assigned before its value can be obtained.

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

## Variable TYPES

There are nine categories of variables: global variables, static variables, instance variables, array elements, value parameters, reference parameters, output parameters, Lazy parameters, and local variables. The sections that follow describe each of these categories.

Example

Public double XF = 4.5;

class TEST  
{  
 public static int x;  
 int y;

void test(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.

## Global Variables

A variable declared outside any class or struct declaration is a global variable. A global variable can be declared with or without enclosing package. The initial value of a global variable is the default value of the variable’s type.

## Instance Variables

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

### Instance variables in classes

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

The initial value of an instance variable of a class is the default value of the variable’s type.

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

### 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.

## 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 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 of the variable’s type.

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

## 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 of the type of the array elements.

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

## value Parameters

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

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) or anonymous function to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter normally ceases to exist upon return of the function member or anonymous function. However, if the value parameter is captured by an anonymous function, its life time 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.

## 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.

* A variable must be definitely assigned before it can be passed as a reference parameter in a function member or delegate invocation.
* Within a function member or anonymous function, a reference parameter is considered initially assigned.

Within an instance method or instance accessor of a struct type, the **this** keyword behaves exactly as a reference parameter of the struct type.

## 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.

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

Within an instance constructor of a struct type, the **this** keyword behaves exactly as an output parameter of the struct type.

## Lazy Parameters

A parameter declared with a yield modifier is a  ***lazy parameter***.

A lazy parameter is an anonymous function that returns the value of the parameter whenever it was invoked within the function member (method, instance constructor, accessor, or operator) or anonymous function to which the parameter belongs. A lazy parameter will be evaluated whenever it’s value is needed, otherwise it will not be evaluated.

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

## Local VARIABLES

A ***local variable*** is declared by a *local-var-decl*, which may occur in a block, a *for-statement*, a *switch-statement* or a *using-statement*; or by a *foreach-statement* or *a specific-catch-clause* for a *try-statement*.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends at least from entry into the block, *for-statement, switch-statement*, *using-statement, foreach-statement*, or *specific-catch-clause* with which it is associated, until execution of that block*, for-statement, switch-statement, using-statement, foreach-statement,* or *specific-catch-clause* ends in any way. (Entering an enclosed block or calling a method suspends, but does not end, execution of the current block, *for-statement, switch-statement, using-statement*, *foreach-statement, or specific-catch-clause*.) If the local variable is captured by an anonymous function, 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 block, *for-statement, switch-statement, using-statement, foreach-statement, or specific-catch-clause* is entered recursively, a new instance of the local variable is created each time, and its local-variable-initializer, if any, is evaluated each time.

A local variable introduced by a local-variable-declaration is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable introduced by a local-variable-declaration is considered initially unassigned. A local-variable-declaration may include a local-variable-initializer, in which case the variable is considered definitely assigned only after the initializing expression.

Within the scope of a local variable introduced by a *local-var-decl*, it is a compile-time error to refer to that local variable in a textual position that precedes its local variable declarator. If the local variable declaration is implicit, it is also an error to refer to the variable within its local variable declarator.

A local variable introduced by a *foreach-statement* or a *specific-catch-clause* is considered definitely assigned in its entire scope.

The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is 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.

## 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.
* For a variable of a reference-type, the default value is null.

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.

## 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

In C and C++, a variable-reference is known as an lvalue.

## atomicity of variable references

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

7

# definite ASSIGNMENTS

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

* An initially assigned variable is always considered definitely assigned.
* An initially unassigned variable 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 in which the variable is the left operand.
* An invocation expression or object creation expression that passes the variable as an output parameter.
* For a local variable, a local variable declaration that includes a variable initializer.

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

## Initially Assigned

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.

## Initially Unassigned

The following categories of variables are classified as initially unassigned:

* Instance variables of initially unassigned struct variables.
* Output parameters, including the **this** variable of struct instance constructors.
* Local variables, except those declared in a catch clause or a foreach statement.

## Rules For Definite Assignments

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

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

* At the beginning of each statement
* At the end point 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 may (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.

## General rules for statements

* v is not definitely assigned at the beginning of a function member body.
* v is definitely assigned at the beginning of any unreachable statement.
* 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.
* The definite assignment state of v at the end point of a block, checked, unchecked, if, while, do, for, foreach, sync, 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.

## 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.

## 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.

## 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).

## 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.

## 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.

## 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.

## Do statements

For a do statement stmt of the form:

do do-body while ( expr ) ;

* v has the same definite assignment state on the control flow transfer from the beginning of stmt to do-body as at the beginning of stmt.
* 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.

## For statements

Definite assignment checking for a for statement of the form:

FOR OPEN\_PARENS for\_initializer? ';' expression? ';' for\_iterator? CLOSE\_PARENS embedded\_statement

is done as if the statement were written:

{  
 for-initializer ;  
 while ( for-condition ) {  
 embedded\_statement;  
 for-iterator ;  
 }  
}

If the for-condition is omitted from the for statement, then evaluation of definite assignment proceeds as if for-condition were replaced with true in the above expansion.

## 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.

## 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.

## 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 must be definitely assigned either:
* after expr
* or at the end of the finally block of a try-finally or try-catch-finally that encloses the return statement.

For a statement stmt of the form:

return ;

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

## 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).

## Try-finally statements

For a try statement stmt of the form:

try try-block finally finally-block

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

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

## Try-catch-finally statements

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

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

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

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

The following example demonstrates how the different blocks of a try statement affect definite assignment.

class EXAMPLE  
{  
 static void TC() {  
 int i, j;  
 try {  
 goto END;  
 // neither i nor j definitely assigned  
 i = 1;  
 // i definitely assigned  
 }

catch(Exception ex) {  
 // 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  
 END:;  
 // j definitely assigned  
  
 }  
}

## Foreach statements

For a foreach statement stmt of the form:

FOREACH OPEN\_PARENS type identifier IN expression CLOSE\_PARENS 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.

## Using statements

For a using statement stmt of the form:

USING OPEN\_PARENS resource\_acquisition CLOSE\_PARENS 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.

## Sync statements

For a sync statement stmt of the form:

sync ( expr ) 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 statement is the same as the state of v at the end of expr.

## 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.

## MIXINS

Mixins are never checked against definite assignments. The mixin block will be checked according to the parent block context.

Mixins can use a variable that was not declared within it but it was defined outside of it, so the compiler must always check the mixin block on demand.

## General rules for simple expressions

The following rule applies to these kinds of expressions: literals, simple names, member access expressions, non-indexed base access expressions, typeof expressions, and default value expressions.

* 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.

## General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions, element access expressions, base access expressions with indexing, increment and decrement expressions, cast expressions, unary +, -, ~, \*, &, ?!, ?~ expressions, binary +, -, \*, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ expressions, compound assignment expressions, checked and unchecked expressions, plus array and delegate creation expressions.

Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed order. For example, the binary % operator evaluates the left hand side of the operator, then the right hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. For an expression expr, which has sub-expressions expr1, expr2, ..., exprn, evaluated in that order:

* The definite assignment state of v at the beginning of expr1 is the same as the definite assignment state at the beginning of expr.
* The definite assignment state of v at the beginning of expri (i greater than one) is the same as the definite assignment state at the end of expri-1.
* The definite assignment state of v at the end of expr is the same as the definite assignment state at the end of exprn.

## Invocation expressions and object creation expressions

For an invocation expression expr of the form:

primary-expression ( arg1 , arg2 , … , argn )

or an object creation expression of the form:

new type ( arg1 , arg2 , … , argn )

* 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 arg1 is the same as the state of v after primary-expression.
* For an object creation expression, the definite assignment state of v before arg1 is the same as the state of v before expr.
* For each argument argi, the definite assignment state of v after argi is determined by the normal expression rules, ignoring any ref or out modifiers.
* For each argument argi for any i greater than one, the definite assignment state of v before argi is the same as the state of v after argi-1.
* If the variable v is passed as an out argument (i.e., an argument of the form “out v”) in any of the arguments, then the state of v after expr is definitely assigned. Otherwise; the state of v after expr is the same as the state of v after argn.
* For array initializers, object initializers, collection initializers and anonymous object initializers, the definite assignment state is determined by the expansion that these constructs are defined in terms of.

## Simple assignment expressions

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

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

## && expressions

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

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

In the example

class A  
{  
 static void F(int x, int y) {  
 int i;  
 if (x >= 0 && (i = y) >= 0) {  
 // 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 being unassigned.

## || expressions

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

* The definite assignment state of v before expr-first is the same as the definite assignment state of v before expr.
* The definite assignment state of v before expr-second is definitely assigned if the state of v after expr-first is either definitely assigned or “definitely assigned after false expression”. Otherwise, it is not definitely assigned.
* The definite assignment statement of v after expr is determined by:
* If expr-first is a constant expression with the value true, then the definite assignment state of v after expr is the same as the definite assignment state of v after expr-first.
* Otherwise, 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.

In the example

class EXMP  
{  
 static void L(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 being unassigned.

## ! expressions

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

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

## ?? 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 with value null, then the 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.

## ?: expressions

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

* The definite assignment state of v before expr-cond is the same as the state of v before expr.
* The definite assignment state of v before expr-true is definitely assigned if and only if one of the following holds:
* expr-cond is a constant expression with the value false
* the state of v after expr-cond is definitely assigned or “definitely assigned after true expression”.
* The definite assignment state of v before expr-false is definitely assigned if and only if one of the following holds:
* expr-cond is a constant expression with the value true
* 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 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 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.

## Anonymous functions

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

* The definite assignment state of an outer variable v before body is the same as the state of v before expr. That is, definite assignment state of outer variables is inherited from the context of the anonymous function.
* The definite assignment state of an outer variable v after expr is the same as the state of v before expr.

The example

delegate bool Filter(int i);

void F() {  
 int max;

// Error, max is not definitely assigned  
 Filter f = (int n) => n < max;

max = 5;  
 DoWork(f);  
}

generates a compile-time error since max is not definitely assigned where the anonymous function is declared. The example

delegate void D();

void F() {  
 int n;  
 D d = () => { n = 1; };

d();

// Error, n is not definitely assigned  
 WriteLn(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.

8

# anonymous functions

***A***n anonymous method expression or lambda expression is classified as an anonymous function. The expression does not have a type but can be implicitly converted to a compatible delegate type. 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.

The examples that follow use a generic delegate type AFunction<A,R> which represents a function that takes an argument of type A and returns a value of type R:

delegate R AFunction <A,R>(A arg);

In the assignments

AFunction <int,int> f1 = x => x + 1; // Ok

AFunction <int,double> f2 = x => x + 1; // Ok

AFunction <double,int> f3 = x => x + 1; // Error

the parameter and return types of each anonymous function are determined from the type of the variable to which the anonymous function is assigned.

The first assignment successfully converts the anonymous function to the delegate type AFunction<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 AFunction<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.

Anonymous functions may influence overload resolution, and participate in type inference.

## anonymous function to delegate types

Conversion of an anonymous function to a delegate type produces a delegate instance which references the anonymous function and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When the delegate is invoked, the body of the anonymous function is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is null, the **this** value of the enclosing function member, or some other object.

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable instances to the same delegate types are permitted (but not required) to return the same delegate instance. The term semantically identical is used here to mean that execution of the anonymous functions will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

delegate double Function(double x);

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

static void F(double[] a, double[] b) {  
 a = Apply(a, (double x) =>x \* 2);  
 b = Apply(b, (double y) =>y \* 2);  
 ...  
 }  
}

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.

## Implementation

This section describes a possible implementation of anonymous function conversions in terms of other V# constructs. The remainder of this section gives several examples of code that contains anonymous functions with different characteristics. For each example, a corresponding translation to code that uses only other V# constructs is provided.

In this example, the anonymous function captures a local variable:

public delegate void D();

int e;

class Test  
{

int x;  
 void F() {  
 int y = 123;  
 D d = () => { y = x + e; };  
 }  
}

This can be translated to a delegate instantiation that references a compiler generated instance method in which the code of the anonymous function is placed:

int e;

class Test  
{

int x;  
 void F() {

int y = 123;  
 Anonymous\_$Locals1 \_\_l1 = new Anonymous\_$Locals1();  
 \_\_l1.y = y;

\_\_l1.\_this = this;  
 D d = new D(\_\_l1.\_\_Method1);  
 }

}

class Anonymous\_$Locals1  
 {

public Test \_this;  
 public int y;

public void \_\_Method1() {  
 y = \_this.x + e;  
 }  
 }

## Method group conversions

An implicit conversion exists from a method group to a compatible delegate type. Given a delegate type D and an expression E that is classified as a method group, an implicit conversion exists from E to D if E contains at least one method that is applicable in its normal form to an argument list constructed by use of the parameter types and modifiers of D, as described in the following.

The compile-time application of a conversion from a method group E to a delegate type D is described in the following. Note that the existence of an implicit conversion from E to D does not guarantee that the compile-time application of the conversion will succeed without error.

* A single method M is selected corresponding to a method invocation of the form E(A), with the following modifications:
* The argument list A is a list of expressions, each classified as a variable and with the type and modifier (ref or out) of the corresponding parameter in the formal-parameter-list of D.
* The candidate methods considered are only those methods that are applicable in their normal form, not those applicable only in their expanded form.
* If the algorithm produces an error, then a compile-time error occurs. Otherwise the algorithm produces a single best method M having the same number of parameters as D and the conversion is considered to exist.
* The selected method M must be compatible with the delegate type D, or otherwise, a compile-time error occurs.
* If the selected method M is an instance method, the instance expression associated with E determines the target object of the delegate.
* If the selected method M is an extension method which is denoted by means of a member access on an instance expression, that instance expression determines the target object of the delegate.
* The result of the conversion is a value of type D, namely a newly created delegate that refers to the selected method and target object.

The following example demonstrates method group conversions:

delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test  
{  
 static string F(object o) {...}

static void G() {  
 D1 d1 = F; // Ok  
 D2 d2 = F; // Ok  
 D3 d3 = F; // Error – not applicable  
 D4 d4 = F; // Error – not applicable in normal form  
 D5 d5 = F; // Error – applicable but not compatible

}  
}

The assignment to d1 implicitly converts the method group F to a value of type D1.

The assignment to d2 shows how it is possible to create a delegate to a method that has less derived (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.

Method groups may influence overload resolution, and participate in type inference.

9

# NAMING CONVENTIONS

***N***ames are used to refer to entities declared in a program. A declared entity is a package, class type (normal or enum), interface type (normal or annotation type), member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor), parameter (to a method, constructor, or exception handler), or local variable. Names in programs are either simple, consisting of a single identifier, or qualified, consisting of a sequence of identifiers separated by "." tokens.

Every declaration that introduces a name has a scope, which is the part of the program text within which the declared entity can be referred to by a simple name. A qualified name N.x may be used to refer to a member of a package or reference type, where N is a simple or qualified name and x is an identifier. If N names a package, then x is a member of that package, which is either a class or interface type or a sub package. If N names a reference type or a variable of a reference type, then x names a member of that type, which is either a class, an interface, a field, or a method. In determining the meaning of a name, the context of the occurrence is used to disambiguate among packages, types, variables, and methods with the same name.

Access control can be specified in a class, interface, method, or field declaration to control when access to a member is allowed. Access is a different concept from scope. Access specifies the part of the program text within which the declared entity can be referred to by a qualified name. Access to a declared entity is also relevant in a field access expression, a method invocation expression in which the method is not specified by a simple name, a method reference expression, or a qualified class instance creation expression. In the absence of an access modifier, most declarations have package access, allowing access anywhere within the package that contains its declaration; other possibilities are public, protected, and private. Fully qualified and canonical names are also discussed in this chapter.

The standard library of V# attempt to use, whenever possible, names chosen according to the conventions presented below. These conventions help to make code more readable and avoid certain kinds of name conflicts.

## Packages

Developers should take steps to avoid the possibility of two published packages having the same name by choosing unique package names for packages that are widely distributed. This allows packages to be easily and automatically installed and catalogued.

Names of packages intended for general purpose use must have the first letter of each word capitalized.

Names of packages intended only for local use should have a first identifier that begins with a lowercase letter.

**Example**: Standard.Input, standard.local.namespace

## Classes

Names of class types should be descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized.

**Example**: OutputStream, MemoryManager

## Interfaces

Names of interface types should be short and descriptive, not overly long, in mixed case with the first letter of each word capitalized. The name may be a descriptive noun or noun phrase, which is appropriate when an interface is used as if it were an abstract superclass, or it may be an adjective describing a behavior.

The first letter of the name must be I.

**Example**: IComparable, IInputManager.

## templates

Template names should be pithy (single character if possible) yet evocative, and should not include lower case letters. This makes it easy to distinguish type parameters from ordinary classes and interfaces.

Template name must be meaningful.

**Example**: For elements we may use (E), For keys we may use (K), For values we may use (V)...

## methods

Method names should be verbs or verb phrases, in mixed case, with the first letter of each word capitalized. Here are some additional specific conventions for method names:

* Property getter and setter methods of a variable named V should have a getter named getV and a setter named setV.
* A method that returns the length of something should be named Length, as in class String.
* A method that tests a boolean condition V about an object should be named isV.
* A method that converts its object to a particular format F should be named toF.
* Event add, remove, raise methods with a name Click should be named addClick, removeClick, raiseClick.

Generally, all compiler generated methods should be verbs or verb phrases, in mixed case, with the first letter lowercase and the first letter of any subsequent words capitalized.

Whenever possible and appropriate, basing the names of methods in a new class on names in an existing class that is similar, especially a class from the Java SE platform API, will make it easier to use.

**Example**: DoWork, addOnClick, getName, isOn…

## fields

Names of fields that are not const should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. Note that well-designed classes have very few public or protected fields, except for fields that are constants (static const fields).

Fields should have names that are nouns, noun phrases, or abbreviations for nouns.

**Example**: totalSize

## constants

The names of constants in interface types should be, and const variables of class types may conventionally be, a sequence of one or more words, acronyms, or abbreviations, all uppercase, with components separated by underscore "\_" characters. Constant names should be descriptive and not unnecessarily abbreviated. Conventionally they may be any appropriate part of speech.

**Example**: MAX\_VALUE, MIN\_VALUE

A group of constants that represent alternative values of a set, or, less frequently, masking bits in an integer value, are sometimes usefully specified with a common acronym as a name prefix.

**Example**: FS\_OPEN, FS\_CLOSED where “FS” stands for **F**ile**S**tate

## Local Variables AND PARAMETERS

Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words, such as:

* Acronyms, that is the first letter of a series of words, as in fs for a variable holding a reference to a FileStream.
* Abbreviations, as in buf holding a pointer to a buffer of some kind
* Mnemonic terms, organized in some way to aid memory and understanding, typically by using a set of local variables with conventional names patterned after the names of parameters to widely used classes. For example:
  + input and output, whenever some kind of input and output are involved, patterned after the fields of System
  + off and len, whenever an offset and length are involved.

One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type.

Conventional one-character names are:

* b for a byte
* c for a char
* d for a double
* e for an Exception
* f for a float
* i, j, and k for int
* l for a long
* o for an Object
* s for a String
* v for an arbitrary value of some type

Local variable or parameter names that consist of only two or three lowercase letters should not conflict with the first component of unique package names.

10

# SCOPE OF DECLARATION

***T***he scope of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name, provided it is visible.

A declaration is said to be in scope at a particular point in a program if and only if the declaration's scope includes that point.

* The scope of the declaration of an observable top level package is all observable compilation units. The declaration of a package that is not observable is never in scope.
* The declaration of a sub package is never in scope.
* The scope of a type imported by a single-type-import declaration or a type import declaration is all the class and interface type declarations in the compilation unit in which the import declaration appears, as well as any annotations on the package declaration (if any) of the compilation unit.
* The scope of a member imported by a single-static-import declaration or a static import declaration is all the class and interface type declarations in the compilation unit in which the import declaration appears, as well as any annotations on the package declaration (if any) of the compilation unit.
* The scope of a top level type is all type declarations in the package in which the top level type is declared. The scope of a declaration of a member m declared in or inherited by a class type C is the entire body of C, including any nested type declarations.
* The scope of a declaration of a member m declared in or inherited by an interface type I is the entire body of I, including any nested type declarations.
* The scope of an enum constant C declared in an enum type T is the body of T, and any case label of a switch statement whose expression is of enum type T.
* The scope of a formal parameter of a method, constructor, or lambda expression is the entire body of the method, constructor, or lambda expression.
* The scope of a class's type parameter is the type parameter section of the class declaration, the type parameter section of any superclass or super interface of the class declaration, and the class body.
* The scope of an interface's type parameter is the type parameter section of the interface declaration, the type parameter section of any super interface of the interface declaration, and the interface body.
* The scope of a method's type parameter is the entire declaration of the method, including the type parameter section, but excluding the method modifiers.
* The scope of a constructor's type parameter is the entire declaration of the constructor, including the type parameter section, but excluding the constructor modifiers.
* The scope of a local class declaration immediately enclosed by a block is the rest of the immediately enclosing block, including its own class declaration.
* The scope of a local class declaration immediately enclosed by a switch block statement group is the rest of the immediately enclosing switch block statement group, including its own class declaration.
* The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement.
* The scope of a local variable declared in the for Init part of a basic for statement includes all of the following:
* Its own initializer
* Any further declarators to the right in the For Init part of the for statement
* The Expression and For Iterator parts of the for statement
* The contained Statement
* The scope of a local variable declared in a foreach statement is the contained Statement.
* The scope of a parameter of an exception handler that is declared in a catch clause of a try statement is the entire block associated with the catch.

## Shadowing

Some declarations may be shadowed in part of their scope by another declaration of the same name, in which case a simple name cannot be used to refer to the declared entity. Shadowing is distinct from hiding, which applies only to members which would otherwise be inherited but are not because of a declaration in a subclass. Shadowing is also distinct from obscuring. A declaration d is said to be visible at point p in a program if the scope of d includes p, and d is not shadowed by any other declaration at p.

When the program point we are discussing is clear from context, we will often simply say that a declaration is visible.

* A declaration d of a type named n shadows the declarations of any other types named n that are in scope at the point where d occurs throughout the scope of d.
* A declaration d of a field or formal parameter named n shadows, throughout the scope of d, the declarations of any other variables named n that are in scope at the point where d occurs.
* A declaration d of a local variable or exception parameter named n shadows, throughout the scope of d, (a) the declarations of any other fields named n that are in scope at the point where d occurs, and (b) the declarations of any other variables named n that are in scope at the point where d occurs but are not declared in the innermost class in which d is declared.
* A declaration d of a method named n shadows the declarations of any other methods named n that are in an enclosing scope at the point where d occurs throughout the scope of d.
* A package declaration never shadows any other declaration.
* A type-import-on-demand declaration never causes any other declaration to be shadowed.
* A static-import-on-demand declaration never causes any other declaration to be shadowed.
* A single-type-import declaration d in a compilation unit c of package p that imports a type named n shadows, throughout c, the declarations of:
* Any top level type named n declared in another compilation unit of p
* Any type named n imported by a type import declaration in c
* Any type named n imported by a static import declaration in c
* A single-static-import declaration d in a compilation unit c of package p that imports a field named n shadows the declaration of any static field named n imported by a static import declaration in c, throughout c.
* A single static import declaration d in a compilation unit c of package p that imports a method named n with signature s shadows the declaration of any static method named n with signature s imported by a static-import-on-demand declaration in c, throughout c.
* A single static import declaration d in a compilation unit c of package p that imports a type named n shadows, throughout c, the declarations of:
* Any static type named n imported by a static-import-on-demand declaration in c;
* Any top level type named n declared in another compilation unit of p;
* Any type named n imported by a type-import-on-demand declaration in c.

**Example:**

class Test {

static int x = 1;

[EntryPoint]

public static void Main(String[] args)

{ int x = 0;

WriteLn("x=" + x);

WriteLn(", Test.x=" + Test.x);

}

}

This program produces the output:

x=0, Test.x=1

* The keyword this can also be used to access a shadowed field x, using the form this.x. Indeed, this idiom typically appears in constructors.

**Example:**

class Pair {

int first, second;

public Pair(int first, int second) {

this.first = first;

this.second = second;

} }

Here, the constructor takes parameters having the same names as the fields to be initialized. This is simpler than having to invent different names for the parameters and is not too confusing in this stylized context. In general, however, it is considered poor style to have local variables with the same names as fields.

## Obscuring

A simple name may occur in contexts where it may potentially be interpreted as the name of a variable, a type, or a package. In these situations, the rules of §6.5 specify that a variable will be chosen in preference to a type, and that a type will be chosen in preference to a package. Thus, it is may sometimes be impossible to refer to a visible type or package declaration via its simple name. We say that such a declaration is obscured.

Obscuring is distinct from shadowing and hiding.

The naming conventions of help reduce obscuring, but if it does occur, here are some notes about what you can do to avoid it.

When package names occur in expressions:

* If a package name is obscured by a field declaration, then import declarations can usually be used to make available the type names declared in that package.
* If a package name is obscured by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

The first component of a package name is normally not easily mistaken for a type name, as a type name normally begins with a single uppercase letter. (The Java programming language does not actually rely on case distinctions to determine whether a name is a package name or a type name.)

Obscuring involving class and interface type names is rare. Names of fields, parameters, and local variables normally do not obscure type names because they conventionally begin with a lowercase letter whereas type names conventionally begin with an uppercase letter.

Method names cannot obscure or be obscured by other names. Obscuring involving field names is rare; however:

* If a field name obscures a package name, then an import declaration can usually be used to make available the type names declared in that package.
* If a field name obscures a type name, then a fully qualified name for the type can be used unless the type name denotes a local class.
* Field names cannot obscure method names.
* If a field name is shadowed by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

Obscuring involving constant names is rare:

* Constant names normally have no lowercase letters, so they will not normally obscure names of packages or types, nor will they normally shadow fields, whose names typically contain at least one lowercase letter.
* Constant names cannot obscure method names, because they are distinguished syntactically.

11

# PACKAGES AND IMPORTS

***P***rograms are organized using packages. Packages 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. Each package has its own set of names for types, which helps to prevent name conflicts. The naming structure for packages is hierarchical. The members of a package are class and interface types, which are declared in compilation units of the package, and sub packages, which may contain compilation units and sub packages of their own.

Import directives are provided to facilitate the use of packages.

## PACKAGE MEMBERS

A *package\_member\_declaration* is either a package\_declaration, *default\_member\_declaration* or a type-declaration.

*package\_member\_declarations*

*: package\_member\_declaration+*

*;*

*package\_member\_declaration*

*: package\_declaration*

*| type\_declaration*

*| attributes? all\_member\_modifiers? (default\_member\_declaration)*

*;*

*default\_member\_declaration*

*: constant\_declaration*

*| typed\_member\_declaration*

*| event\_declaration*

*| conversion\_operator\_declarator (body | right\_arrow expression ';')*

*| VOID method\_declaration*

*| mixin\_definition*

*;*

*type\_declaration*

*: attributes? all\_member\_modifiers?*

*(class\_definition | struct\_definition | union\_definition | interface\_definition | enum\_definition | delegate\_definition)*

*;*

A compilation unit or a package body can contain package\_member\_declarations, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or package body.

## Compilation Units

A compilation\_unit is the goal symbol for the syntactic grammar and it defines the overall structure of a source file. A compilation unit consists of zero or more import\_directives followed by zero or more global\_attribute\_section followed by zero or more *package*\_member\_declarations.

*compilation\_unit*

*: BYTE\_ORDER\_MARK? import\_directives?*

*global\_attribute\_section\* package\_member\_declarations? EOF*

*;*

A program consists of one or more compilation units, each contained in a separate source file. When a V# 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 import\_directives of a compilation unit affect the global\_attribute\_section and package\_member\_declarations of that compilation unit, but have no effect on other compilation units.

The global\_attribute\_section of a compilation unit permit the specification of attributes for the target output. Output act as physical containers for types and members.

The package\_member\_declarations of each compilation unit of a program contribute members to a single declaration space called the default package. For example:

File Test1.vs:

class Test1 {}

File Test2.vs:

class Test2 {}

The two compilation units contribute to the single global package, in this case declaring two classes with the fully qualified names Test1 and Test2. 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.

## PACKAGE Declaration

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs. A package\_declaration consists of the keyword package, followed by a package name and body, optionally followed by a semicolon.

*package\_declaration*

*: PACKAGE qi=qualified\_identifier package\_body ';'?*

*;*

*qualified\_identifier*

*: identifier ( '.' identifier )\**

*;*

*package\_body*

*: OPEN\_BRACE import\_directives? package\_member\_declarations? CLOSE\_BRACE*

*;*

A package\_declaration may occur as a top-level declaration in a compilation\_unit or as a member declaration within another package\_declaration. When a package\_declaration occurs as a top-level declaration in a compilation\_unit, the package becomes a member of the default package. When a package\_declaration occurs within another package\_declaration, the inner package becomes a member of the outer package. In either case, the name of a package must be unique within the containing package.

Packages are implicitly public and the declaration of a package cannot include any access modifiers.

Within a package\_body, the optional import\_directives import the names of other package and types, allowing them to be referenced directly instead of through qualified names. The optional package\_member-declarations contribute members to the declaration space of the package. Note that all import\_directives must appear before any member declarations.

The qualified\_identifier of a package\_declaration may be a single identifier or a sequence of identifiers separated by “.” tokens. The latter form permits a program to define a nested package without lexically nesting several package declarations.

## IMPORT DIRECTIVES

An import declaration allows a named type or a static member to be referred to by a simple name that consists of a single identifier. Without the use of an appropriate import declaration, the only way to refer to a type declared in another package, or a static member of another type, is to use a fully qualified name. Import directives impact the name resolution process of *package\_or\_type\_name* and *simple\_name*, but unlike declarations, import directives do not contribute new members to the underlying declaration spaces of the compilation units or packages within which they are used.

*import\_directives*

*: import\_directive+*

*;*

*import\_directive*

*: IMPORT identifier '=' package\_or\_type\_name ';'*

*| IMPORT package\_or\_type\_name ';'*

*| IMPORT STATIC package\_or\_type\_name ';'*

*;*

* + A type import declaration imports a single named type, by mentioning its canonical name.
  + A static import declaration imports all accessible static members with a given name from a type, by giving its canonical name.
  + A package import declaration imports all accessible types with a given name from a package, by giving its canonical name.
  + The scope of an import\_directive extends over the package\_member\_declarations of its immediately containing compilation unit or package body.
  + The scope of an import\_directive specifically does not include its peer using-directives. Thus, peer import\_directives do not affect each other, and the order in which they are written is insignificant.

## Import Static

A static import declaration imports all accessible static members with a given simple name from a type. This makes these static members available under their simple name in the class and interface declarations of the compilation unit in which the single-static-import declaration appears.

It is a compile-time error if the named type is not accessible.

The name must be qualified, or a compile-time error occurs.

**Example:**

package P1  
{  
 public class A {

static void X(){}

public static void Y(){}

}  
}

package P2  
{

import static P1.A;

class A {

self(){

X(); // Error: X is inaccessible due to it’s protection level

Y(); // OK

}

}

}

## Import alias

An ***import alias directive*** introduces an identifier that serves as an alias for a namespace or type within the immediately enclosing compilation unit or namespace body.

Within member declarations in a compilation unit or namespace body that contains an ***import alias directive***, the identifier introduced by the ***import alias directive*** can be used to reference the given namespace or type.

The identifier of an import alias directive must be unique within the declaration space of the compilation unit or namespace that immediately contains the using-alias-directive.

**Example:**

package P1  
{  
 class A {}  
}

namespace P2  
{  
 class A {}  
}

namespace P3  
{  
 import P1;

import P2;

import X = P1.A;

class B: A {} // Error, A is ambiguous

class C: X {} // OK  
}

## PACKAGE ALIAS QUALIFIER

The package alias qualifier :: makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The package 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.

qualified\_alias\_member

: identifier '::' identifier type\_argument\_list?

;

* If the global namespace contains a non-generic type, then the qualified\_alias\_member refers to that type.
* If the global namespace contains a type that has type parameters, then the qualified\_alias\_member refers to that type constructed with the given type arguments.
* The qualified\_alias\_member is undefined and a compile-time error occurs.

Using the package alias qualifier with an alias that references a type causes a compile-time error.

**Example:**

namespace P  
{  
 public class X {}

public class Y {}  
}

namespace P  
{  
 using X = Std;

class A  
 {  
 X.String s1; // Error, X is ambiguous

X::String s2; // Ok  
 }  
}

12

# CLASSES

***A*** class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a super class.

## Class declarations

A class\_definition is a type\_declaration that declares a new class.

type\_declaration

: attributes? all\_member\_modifiers?

(class\_definition | struct\_definition | union\_definition | interface\_definition | enum\_definition | delegate\_definition)

;

class\_definition

: CLASS identifier type\_parameter\_list? class\_base? type\_parameter\_constraints\_clauses?

class\_body ';'?

;

A class\_definition consists of an optional set of attributes, followed by an optional set of all\_member\_modifiers, followed by an optional partial modifier, followed by the keyword class and an identifier that names the class, followed by an optional type\_parameter\_list, followed by an optional class\_base specification, followed by an optional set of type\_parameter\_constraints\_clauses, followed by a class\_body, optionally followed by a semicolon.

A class declaration cannot supply type\_parameter\_constraints\_clauses unless it also supplies a type\_parameter\_list.

A class declaration that supplies a type\_parameter\_list is a generic class declaration. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration, since type parameters for the containing type must be supplied to create a constructed type.

## Class modifiers

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. It is a compile-time error for the new modifier to appear on a class declaration that is not a nested class declaration.

The public, protected, internal, and private modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted.

The abstract, sealed and static modifiers are discussed in the following sections.

### 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 super 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 must include actual implementations of all inherited abstract members, thereby overriding those abstract members.

### 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 super class of another class.

A sealed class cannot also be an abstract class.

The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain 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.

### Static classes

The static modifier is used to mark the class being declared as a static class. A static class cannot be instantiated, cannot be used as a type and can contain only static members. Only a static class can contain declarations of extension methods.

A static class declaration is subject to the following restrictions:

* A static class may not include a sealed or abstract modifier. Note, however, that since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.
* A static class may not include a class\_base specification and cannot explicitly specify a super class or a list of implemented interfaces. A static class implicitly inherits from type object.
* A static class can only contain static members. Note that constants and nested types are classified as static members.
* A static class cannot have members with protected or protected internal declared accessibility.

It is a compile-time error to violate any of these restrictions.

A static class has no instance constructors. It is not possible to declare an instance constructor in a static class, and no default instance constructor is provided for a static class.

The members of a static class are not automatically static, and the member declarations must explicitly include a static modifier (except for constants and nested types). When a class is nested within a static outer class, the nested class is not a static class unless it explicitly includes a static modifier.

### Partial modifier

The partial modifier is used to indicate that this class\_definition is a partial type declaration. Multiple partial type declarations with the same name within an enclosing package or type declaration combine to form one type declaration, following the rules specified in.

Having the declaration of a class distributed over separate segments of program text can be useful if these segments are produced or maintained in different contexts. For instance, one part of a class declaration may be machine generated, whereas the other is manually authored. Textual separation of the two prevents updates by one from conflicting with updates by the other.

## Type parameters

A type parameter is a simple identifier that denotes a placeholder for a type argument supplied to create a constructed type. A type parameter is a formal placeholder for a type that will be supplied later. By contrast, a type argument is the actual type that is substituted for the type parameter when a constructed type is created.

*type\_parameter\_list*

*: '<' type\_parameter (',' type\_parameter)\* '>'*

*;*

*type\_parameter*

*: attributes? identifier*

*;*

Each type parameter in a class declaration defines a name in the declaration space of that class. Thus, it cannot have the same name as another type parameter or a member declared in that class. A type parameter cannot have the same name as the type itself.

## SUPERCLASS specification

A class declaration may include a class\_base specification, which defines the direct super class of the class and the interfaces directly implemented by the class.

*class\_base*

*: ':' class\_type (',' package\_or\_type\_name)\**

*;*

*class\_type*

*: package\_or\_type\_name*

*| OBJECT*

*| STRING*

*;*

*package\_or\_type\_name*

*: (identifier type\_argument\_list? | qualified\_alias\_member) ('.' identifier type\_argument\_list?)\**

*;*

The super class specified in a class declaration can be a constructed class type. A super class cannot be a type parameter on its own, though it can involve the type parameters that are in scope.

class Extend<V> : V {} // Error, type parameter used as super class

### Super classes

When a class\_type is included in the class\_base, it specifies the direct super class of the class being declared. If a class declaration has no class\_base, or if the class\_base lists only interface types, the direct super class is assumed to be object. A class inherits members from its direct super class, as described in.

**Example:**

class X {}

class Y: X {}

class X is said to be the direct super class of Y, and Y is said to be derived from X. Since X does not explicitly specify a direct super class, its direct super class is implicitly object.

For a constructed class type, if a super class is specified in the generic class declaration, the super class of the constructed type is obtained by substituting, for each type\_parameter in the super class declaration, the corresponding type\_argument of the constructed type. Given the generic class declarations

class B<U,V> {...}

class G<T>: B<string,T[]> {...}

the super class of the constructed type G<int> would be B<string,int[]>.

The direct super class of a class type must be at least as accessible as the class type itself.

It is a compile-time error for a public class to derive from a private or internal class.

Furthermore, a generic class declaration cannot use Std.Attribute as a direct or indirect super class.

The super classes of a class type are the direct super class and its super classes. In other words, the set of super classes is the transitive closure of the direct super class relationship.

The object class has no direct super class and is the ultimate super class of all other classes.

When a class B derives from a class A, it is a compile-time error for A to depend on B. A class directly depends on its direct super class (if any) and directly depends on the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the reflexive and transitive closure of the directly depends on relationship.

**Example:**

class A: A {}

Is erroneous because the class depends on itself.

**Example:**

class A: B {}

class B: C {}

class C: A {}

Is in error because the classes circularly depend on themselves.

**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 super class), which depends on B (its immediately enclosing class), which circularly depends on A.

Note that a class does not depend on the classes that are nested within it.

**Example:**

class A  
{  
 class B: A {}  
}

B depends on A (because A is both its direct super class and its immediately enclosing class), but A does not depend on B (since B is neither a super class nor an enclosing class of A). Thus, the example is valid.

It is not possible to derive from a sealed class. In the example

sealed class A {}

class B: A {} // Error, cannot derive from a sealed class

### Interface implementations

A class\_base specification may include a list of interface types, in which case the class is said to directly implement the given interface types.

## 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\_clause*

*: WHERE identifier ':' type\_parameter\_constraints*

*;*

*type\_parameter\_constraints*

*: primary\_constraint (',' secondary\_constraints)?*

*;*

*primary\_constraint*

*: class\_type*

*| CLASS*

*| STRUCT*

*;*

*secondary\_constraints*

*: package\_or\_type\_name (',' package\_or\_type\_name)\**

*;*

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 and one or more secondary constraints.

A primary constraint can be a class type or the reference type constraint class or the value type constraint struct. A secondary constraint can be a type\_parameter or interface\_type.

The reference type constraint specifies that a type argument used for the type parameter must be a reference type. All class types, interface types, delegate types, array types, and type parameters known to be a reference type (as defined below) satisfy this constraint.

The value type constraint specifies that a type argument used for the type parameter must be a non-nullable value type. All non-nullable struct types, enum types, and type parameters having the value type constraint satisfy this constraint. Note that although classified as a value type, a nullable type does not satisfy the value type constraint.

Pointer types are never allowed to be type arguments and are not considered to satisfy either the reference type or value type constraints.

If a constraint is a class type, an interface type, or a type parameter, that type specifies a minimal “base type” that every type argument used for that type parameter must support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile-time. The type argument supplied must satisfy the conditions described in section.

A class\_type constraint must satisfy the following rules:

* The type must be a class type.
* The type must not be sealed.
* The type must not be one of the following types: Std.Array, Std.Delegate, Std.Enum.
* The type must not be object. Because all types derive from object, such a constraint would have no effect if it were permitted.
* At most one constraint for a given type parameter can be a class type.

A type specified as an interface\_type constraint must satisfy the following rules:

* The type must be an interface type.
* A type must not be specified more than once in a given where clause.

In either case, the constraint can involve any of the type parameters of the associated type or method declaration as part of a constructed type, and can involve the type being declared.

Any class or interface type specified as a type parameter constraint must be at least as accessible as the generic type or method being declared.

A type specified as a type\_parameter constraint must satisfy the following rules:

* The type must be a type parameter.
* A type must not be specified more than once in a given where clause.

In addition, there must be no cycles in the dependency graph of type parameters, where dependency is a transitive relation defined 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 must be consistent among dependent type parameters. If type parameter S depends on type parameter T then:

* If S has the value type constraint, then T must not have a class\_type constraint.
* If S also depends on type parameter U and U has a class\_type constraint A and T has a class\_type constraint B then there must be an identity conversion or implicit reference conversion from A to B or an implicit reference conversion from B to A.

**Example:** Use of constraints

interface IComparable<T>  
{  
 int CompareTo(T value);  
}

class SortedList<T> where T: IComparable<T> {...}

**Examples:**

class Circular<S,T>  
 where S: T  
 where T: S // Error, circularity in dependency graph  
{  
 ...  
}

class Sealed<S,T>  
 where S: T  
 where T: struct // Error, T is sealed  
{  
 ...  
}

class A {...}

class B {...}

class Incompat<S,T>  
 where S: A, T  
 where T: B // Error, incompatible class-type constraints  
{  
 ...  
}

class StructWithClass<S,T,U>  
 where S: struct, T  
 where T: U  
 where U: A // Error, A incompatible with struct  
{  
 ...  
}

The effective super class of a type parameter T is defined as follows:

* If T has no primary constraints or type parameter constraints, its effective super class is object.
* If T has a class\_type constraint C but no type-parameter constraints, its effective super class is C.
* If T has the reference type constraint but no class\_type constraints, its effective super class is object.

For the purpose of these rules, if T has a constraint V that is a value-type, use instead the most specific base type of V that is a class-type. This can never happen in an explicitly given constraint, but may occur when the constraints of a generic method are implicitly inherited by an overriding method declaration or an explicit implementation of an interface method.

These rules ensure that the effective super class is always a class-type.

A type parameter is known to be a reference type if it has the reference type constraint.

Values of a constrained type parameter type can be used to access the instance members implied by the constraints.

**Exemple:**

interface IComparable<T>  
{  
 int CompareTo(T value) ;

}

class SortedList<T> where T: IComparable<T>  
{  
 void Compare(T x,T y) {  
 x.CompareTo(y) ;  
 }  
}

the methods of IComparable<T> can be invoked directly on x because T is constrained to always implement IComparable.

## Class body

The class\_body of a class defines the members of that class.

*class\_body*

*: OPEN\_BRACE class\_member\_declarations? CLOSE\_BRACE*

*;*

## Partial types

A type declaration can be split across multiple partial type declarations. The type declaration is constructed from its parts by following the rules in this section, whereupon it is treated as a single declaration during the remainder of the compile-time and run-time processing of the program.

A class\_declaration, struct\_declaration or interface\_declaration represents a partial type declaration if it includes a partial modifier. partial is not a keyword, and only acts as a modifier if it appears immediately before one of the keywords class, struct or interface in a type declaration, or before the type void in a method declaration. In other contexts, it can be used as a normal identifier.

Each part of a partial type declaration must include a partial modifier. It must have the same name and be declared in the same namespace or type declaration as the other parts. The partial modifier indicates that additional parts of the type declaration may exist elsewhere, but the existence of such additional parts is not a requirement; it is valid for a type with a single declaration to include the partial modifier.

All parts of a partial type must be compiled together such that the parts can be merged at compile-time into a single type declaration. Partial types specifically do not allow already compiled types to be extended.

Nested types may be declared in multiple parts by using the partial modifier. Typically, the containing type is declared using partial as well, and each part of the nested type is declared in a different part of the containing type.

The partial modifier is not permitted on delegate or enum declarations.

## Attributes

The attributes of a partial type are determined by combining, in an unspecified order, the attributes of each of the parts. If an attribute is placed on multiple parts, it is equivalent to specifying the attribute multiple times on the type.

## Modifiers

When a partial type declaration includes an accessibility specification (the public, protected, internal, and private modifiers) it must agree with all other parts that include an accessibility specification.

* If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility.
* If one or more partial declarations of a nested type include a new modifier, no warning is reported if the nested type hides an inherited member.
* If one or more partial declarations of a class include an abstract modifier, the class is considered abstract. Otherwise, the class is considered non-abstract.
* If one or more partial declarations of a class include a sealed modifier, the class is considered sealed. Otherwise, the class is considered unsealed.

Note that a class cannot be both abstract and sealed.

When the unsafe modifier is used on a partial type declaration, only that particular part is considered an unsafe context.

## Type parameters and constraints

If a generic type is declared in multiple parts, each part must state the type parameters. Each part must have the same number of type parameters, and the same name for each type parameter, in order.

When a partial generic type declaration includes constraints (where clauses), the constraints must agree with all other parts that include constraints. Specifically, each part that includes constraints must have constraints for the same set of type parameters, and for each type parameter the sets of primary and secondary constraints must be equivalent. Two sets of constraints are equivalent if they contain the same members. If no part of a partial generic type specifies type parameter constraints, the type parameters are considered unconstrained.

The example

partial class Dictionary<K,V>  
 where K: IComparable<K>  
 where V: IKeyProvider<K>, IPersistable  
{  
 ...  
}

partial class Dictionary<K,V>  
 where V: IPersistable, IKeyProvider<K>  
 where K: IComparable<K>  
{  
 ...  
}

partial class Dictionary<K,V>  
{  
 ...  
}

is correct because those parts that include constraints (the first two) effectively specify the same set of primary, secondary, and constructor constraints for the same set of type parameters, respectively.

## Super class

When a partial class declaration includes a super class specification it must agree with all other parts that include a super class specification. If no part of a partial class includes a super class specification, the super class becomes Std.Object.

## Base interfaces

The set of base interfaces for a type declared in multiple parts is the union of the base interfaces specified on each part. A particular base interface may only be named once on each part, but it is permitted for multiple parts to name the same base interface(s). There must only be one implementation of the members of any given base interface.

In the example

partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}

the set of base interfaces for class C is IA, IB, and IC.

Typically, each part provides an implementation of the interface(s) declared on that part; however, this is not a requirement. A part may provide the implementation for an interface declared on a different part:

partial class X  
{  
 int IComparable.CompareTo(object o) {...}  
}

partial class X: IComparable  
{  
 ...  
}

## Members

With the exception of partial methods, the set of members of a type declared in multiple parts is simply the union of the set of members declared in each part. The bodies of all parts of the type declaration share the same declaration space, and the scope of each member extends to the bodies of all the parts. The accessibility domain of any member always includes all the parts of the enclosing type; a private member declared in one part is freely accessible from another part. It is a compile-time error to declare the same member in more than one part of the type, unless that member is a type with the partial modifier.

partial class A  
{  
 int x; // Error, cannot declare x more than once

partial class Inner // Ok, Inner is a partial type  
 {  
 int y;  
 }  
}

partial class A  
{  
 int x; // Error, cannot declare x more than once

partial class Inner // Ok, Inner is a partial type  
 {  
 int z;  
 }  
}

The ordering of members within a type is rarely significant to C# code, but may be significant when interfacing with other languages and environments. In these cases, the ordering of members within a type declared in multiple parts is undefined.

## Partial methods

Partial methods can be defined in one part of a type declaration and implemented in another. The implementation is optional; if no part implements the partial method, the partial method declaration and all calls to it are removed from the type declaration resulting from the combination of the parts.

Partial methods cannot define access modifiers, but are implicitly private. Their return type must be void, and their parameters cannot have the out modifier. The identifier partial is recognized as a special keyword in a method declaration only if it appears right before the void type; otherwise it can be used as a normal identifier. A partial method cannot explicitly implement interface methods.

There are two kinds of partial method declarations: If the body of the method declaration is a semicolon, the declaration is said to be a defining partial method declaration. If the body is given as a block, the declaration is said to be an implementing partial method declaration. Across the parts of a type declaration there can be only one defining partial method declaration with a given signature, and there can be only one implementing partial method declaration with a given signature. If an implementing partial method declaration is given, a corresponding defining partial method declaration must exist, and the declarations must match as specified in the following:

* The declarations must have the same modifiers (although not necessarily in the same order), method name, number of type parameters and number of parameters.
* Corresponding parameters in the declarations must have the same modifiers (although not necessarily in the same order) and the same types (modulo differences in type parameter names).
* Corresponding type parameters in the declarations must have the same constraints (modulo differences in type parameter names).

An implementing partial method declaration can appear in the same part as the corresponding defining partial method declaration.

Only a defining partial method participates in overload resolution. Thus, whether or not an implementing declaration is given, invocation expressions may resolve to invocations of the partial method. Because a partial method always returns void, such invocation expressions will always be expression statements. Furthermore, because a partial method is implicitly private, such statements will always occur within one of the parts of the type declaration within which the partial method is declared.

If no part of a partial type declaration contains an implementing declaration for a given partial method, any expression statement invoking it is simply removed from the combined type declaration. Thus the invocation expression, including any constituent expressions, has no effect at run-time. The partial method itself is also removed and will not be a member of the combined type declaration.

If an implementing declaration exist for a given partial method, the invocations of the partial methods are retained. The partial method gives rise to a method declaration similar to the implementing partial method declaration except for the following:

* The partial modifier is not included
* The attributes in the resulting method declaration are the combined attributes of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.
* The attributes on the parameters of the resulting method declaration are the combined attributes of the corresponding parameters of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.

If a defining declaration but not an implementing declaration is given for a partial method M, the following restrictions apply:

* It is a compile-time error to create a delegate to method.
* It is a compile-time error to refer to M inside an anonymous function that is converted to an expression tree type.
* Expressions occurring as part of an invocation of M do not affect the definite assignment state, which can potentially lead to compile-time errors.
* M cannot be the entry point for an application.

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);  
 }  
}

## Name binding

Although each part of an extensible type must be declared within the same namespace, the parts are typically written within different namespace declarations. Thus, different using directives may be present for each part. When interpreting simple names within one part, only the import directives of the namespace declaration(s) enclosing that part are considered. This may result in the same identifier having different meanings in different parts:

namespace N  
{  
 using List = System.Collections.ArrayList;

partial class A  
 {  
 List x; // x has type System.Collections.ArrayList  
 }  
}

namespace N  
{  
 using List = Widgets.LinkedList;

partial class A  
 {  
 List y; // y has type Widgets.LinkedList  
 }  
}

## Class members

The members of a class consist of the members introduced by its class-member-declarations and the members inherited from the direct super class.

*class\_member\_declarations*

*: class\_member\_declaration+*

*;*

*class\_member\_declaration*

*: attributes? all\_member\_modifiers? (common\_member\_declaration | destructor\_definition)*

*;*

*common\_member\_declaration*

*: constant\_declaration*

*| typed\_member\_declaration*

*| event\_declaration*

*| conversion\_operator\_declarator (body | right\_arrow expression ';')*

*| constructor\_declaration*

*| VOID method\_declaration*

*| class\_definition*

*| struct\_definition*

*| interface\_definition*

*| enum\_definition*

*| mixin\_definition*

*| delegate\_definition*

*;*

The members of a class type are divided into the following categories:

* Constants, which represent constant values associated with the class.
* Fields, which are the variables of the class.
* Methods, which implement the computations and actions that can be performed by the class.
* Properties, which define named characteristics and the actions associated with reading and writing those characteristics.
* Events, which define notifications that can be generated by the class.
* Indexers, which permit instances of the class to be indexed in the same way (syntactically) as arrays.
* Operators, which define the expression operators that can be applied to instances of the class.
* Instance constructors, which implement the actions required to initialize instances of the class
* Destructors, which implement the actions to be performed before instances of the class are permanently discarded.
* Static constructors, which implement the actions required to initialize the class itself.
* Types, which represent the types that are local to the class.

Members that can contain executable code are collectively known as the function members of the class type. The function members of a class type are the methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors of that class type.

A class\_declaration creates a new declaration space, 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, destructors and static constructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
* The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
* The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by ref and out.
* The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class, and two constructors declared in the same class may not have signatures that differ solely by ref and out.
* The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
* The signature of an operator must differ from the signatures of all other operators declared in the same class.

The inherited members of a class type are not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

## The instance type

Each class declaration has an associated bound type, the instance type. For a generic class declaration, the instance type is formed by creating a constructed type 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. The following shows several class declarations along with their instance types:

class A<T> // instance type: A<T>  
{  
 class B {} // instance type: A<T>.B

class C<U> {} // instance type: A<T>.C<U>  
}

class D {} // instance type: D

## Members of constructed types

The non-inherited members of a constructed type are obtained by substituting, for each type\_parameter in the member declaration, the corresponding type\_argument of the constructed type. The substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution.

For example, given the generic class declaration

class Gen<T,U>  
{  
 public T[,] a;

public void G(int i, T t, Gen<U,T> gt) {...}

public U Prop { get {...} set {...} }

public int H(double d) {...}  
}

the constructed type Gen<int[],IComparable<string>> has the following members:

public int[,][] a;

public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}

public IComparable<string> Prop { get {...} set {...} }

public int H(double d) {...}

The type of the member a in the generic class declaration Gen is “two-dimensional array of T”, so the type of the member a in the constructed type above is “two-dimensional array of one-dimensional array of int”, or int[,][].

Within instance function members, the type of this is the instance type 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 is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example:

class C<V>  
{  
 public V f1;  
 public C<V> f2 = null;

public C(V x) {  
 this.f1 = x;  
 this.f2 = this;  
 }  
}

class Application  
{  
 static void Main() {  
 C<int> x1 = new C<int>(1);  
 Writeln(x1.f1); // Prints 1

C<double> x2 = new C<double>(3.1415);  
 Writeln(x2.f1); // Prints 3.1415  
 }  
}

## Inheritance

A class inherits the members of its direct super class type. Inheritance means that a class implicitly contains all members of its direct super class type, except for the instance constructors, destructors and static constructors of the super 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 super class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
* Instance constructors, destructors, and static constructors are not inherited, but all other members are, regardless of their declared accessibility. However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
* A derived class can hide inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove that member—it merely makes that member inaccessible directly through the derived class.
* An instance of a class contains a set of all instance fields declared in the class and its super classes, and an implicit conversion exists from a derived class type to any of its super 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 super classes.
* A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which that function member is invoked.

The inherited member of a constructed class type are the members of the immediate super class type, which is found by substituting the type arguments of the constructed type for each occurrence of the corresponding type parameters in the base\_class\_specification. These members, in turn, are transformed by substituting, for each type\_parameter in the member declaration, the corresponding type\_argument of the base\_class\_specification.

class B<U>  
{  
 public U F(long index) {...}  
}

class D<T>: B<T[]>  
{  
 public T G(string s) {...}  
}

In the above example, the constructed type D<int> has a non-inherited member public int G(string s) obtained by substituting the type argument int for the type parameter T. D<int> also has an inherited member from the class declaration B. This inherited member is determined by first determining the super class type B<int[]> of D<int> by substituting int for T in the super 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).

## 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 super class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a new modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in.

If a new modifier is included in a declaration that doesn’t hide an inherited member, a warning to that effect is issued. This warning is suppressed by removing the new modifier.

## Access modifiers

A class\_member\_declaration can have any one of the five possible kinds of declared accessibility: 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.

## Constituent types

Types that are used in the declaration of a member are called the constituent types of that member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or instance constructor. The constituent types of a member must be at least as accessible as that member itself.

## Static and instance members

Members of a class are either static members or instance members. Generally speaking, it is useful to think of static members as belonging to class types and instance members as belonging to objects (instances of class types).

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 of the form E.M, E must denote a type containing M. It is a compile-time error for E to denote an instance.
* A static field identifies exactly one storage location to be shared by all instances of a given closed class type. No matter how many instances of a given closed class type are created, there is only ever one copy of a static field.
* A static function member (method, property, event, operator, or constructor) does not operate on a specific instance, and it is a compile-time error to refer to this in such a function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a static modifier, it declares an instance member. (An instance member is sometimes called a non-static member.) Instance members have the following characteristics:

* When an instance member M is referenced in a member\_access of the form E.M, E must denote an instance of a type containing M. It is a binding-time error for E to denote a type.
* Every instance of a class contains a separate set of all instance fields of the class.
* An instance function member (method, property, indexer, instance constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as this.

The following example illustrates the rules for accessing static and instance members:

class Test  
{  
 int x;  
 static int y;

void F() {  
 x = 1; // Ok, same as this.x = 1  
 y = 1; // Ok, same as Test.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; // Ok  
 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 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, instance members must be accessed through instances, and static members must be accessed through types.

## Nested types

A type declared within a class or struct declaration is called a nested type. A type that is declared within a compilation unit or namespace is called a non-nested type.

In the example

using System;

class A  
{  
 class B  
 {  
 static void F() {  
 Writeln("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.

### Fully qualified name

The fully qualified name for a nested type is S.N where S is the fully qualified name of the type in which type N is declared.

### 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.

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.

### Hiding

A nested type may hide a base member. The new modifier is permitted on nested type declarations so that hiding can be expressed explicitly. The example

using System;

class Base  
{  
 public static void M() {  
 Writeln("Base.M");  
 }  
}

class Derived: Base   
{  
 new public class M   
 {  
 public static void F() {  
 Writeln("Derived.M.F");  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 Derived.M.F();  
 }  
}

shows a nested class M that hides the method M defined in Base.

### this access

A nested type and its containing type do not have a special relationship with regard to this-access. 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. The following example

using System;

class C  
{  
 int i = 123;

public void F() {  
 Nested n = new Nested(this);  
 n.G();  
 }

public class Nested  
 {  
 C this\_c;

public Nested(C c) {  
 this\_c = c;  
 }

public void G() {  
 Writeln(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.

### Access to private and protected members of the containing type

A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have private and protected declared accessibility. The example

using System;

class C   
{  
 private static void F() {  
 Writeln("C.F");  
 }

public class Nested   
 {  
 public static void G() {  
 F();  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 C.Nested.G();  
 }  
}

shows a class C that contains a nested class Nested. Within Nested, the method G calls the static method F defined in C, and F has private declared accessibility.

A nested type also may access protected members defined in a base type of its containing type. In the example

using System;

class Base   
{  
 protected void F() {  
 Writeln("Base.F");  
 }  
}

class Derived: Base   
{  
 public class Nested   
 {  
 public void G() {  
 Derived d = new Derived();  
 d.F(); // ok  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 Derived.Nested n = new Derived.Nested();  
 n.G();  
 }  
}

the nested class Derived.Nested accesses the protected method F defined in Derived’s super class, Base, by calling through an instance of Derived.

### Nested types in generic classes

A generic class declaration can contain nested type declarations. The type parameters of the enclosing class can be used within the nested types. A nested type declaration can contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, must be named. However, from within the outer class, the nested type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the nested type. The following example shows three different correct ways to refer to a constructed type created from Inner; the first two are equivalent:

class Outer<T>  
{  
 class Inner<U>  
 {  
 public static void F(T t, U u) {...}  
 }

static void F(T t) {  
 Outer<T>.Inner<string>.F(t, "abc"); // These two statements have  
 Inner<string>.F(t, "abc"); // 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  
 }  
}

Although it is bad programming style, a type parameter in a nested type can hide a member or type parameter declared in the outer type:

class Outer<T>  
{  
 class Inner<T> // Valid, hides Outer’s T  
 {  
 public T t; // Refers to Inner’s T  
 }  
}

## Reserved member names

To facilitate the underlying C# run-time implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying 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, and can be hidden with the new modifier.

The reservation of these names serves three purposes:

* To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
* To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
* 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.

The declaration of a destructor also causes a signature to be reserved.

### Member names reserved for properties

For a property P (§10.7) of type T, the following signatures are reserved:

T get\_P();  
void set\_P(T value);

Both signatures are reserved, even if the property is read-only or write-only.

In the example

using System;

class A  
{  
 public int P {  
 get { return 123; }  
 }  
}

class B: A  
{  
 new public int get\_P() {  
 return 456;  
 }

new public void set\_P(int value) {  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 A a = b;  
 Writeln(a.P);  
 Writeln(b.P);  
 Writeln(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

### Member names reserved for events

For an event E of delegate type T, the following signatures are reserved:

void add\_E(T handler);  
void remove\_E(T handler);

### Member names reserved for indexers

For an indexer 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.

### Member names reserved for destructors

For a class containing a destructor, the following signature is reserved:

void Finalize();

## Constants

A constant is a class member that represents a constant value: a value that can be computed at compile-time. A constant-declaration introduces one or more constants of a given type.

constant-declaration:  
attributesopt constant-modifiersopt 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 (§17), a new modifier (§10.3.4), and a valid combination of the four access modifiers (§10.3.5). The attributes and modifiers apply to all of the members declared by the constant-declaration. Even though constants are considered static members, a constant-declaration neither requires nor allows a static modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The type of a constant-declaration specifies the type of the members introduced by the declaration. The type is followed by a list of constant-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 (§7.19) that gives the value of the member.

The type specified in a constant declaration must 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 must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion (§6.1).

The type of a constant must be at least as accessible as the constant itself (§3.5.4).

The value of a constant is obtained in an expression using a simple-name (§7.6.2) or a member-access (§7.6.4).

A constant can itself participate in a constant-expression. Thus, a constant may be used in any construct that requires a constant-expression. Examples of such constructs include case labels, goto case statements, enum member declarations, attributes, and other constant declarations.

As described in §7.19, a constant-expression is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a reference-type other than string is to apply the new operator, and since the new operator is not permitted in a constant-expression, the only possible value for constants of reference-types other than string is null.

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile-time by a constant-expression, a readonly field (§10.5.2) may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example

class A  
{  
 public const double X = 1.0, Y = 2.0, Z = 3.0;  
}

is equivalent to

class A  
{  
 public const double X = 1.0;  
 public const double Y = 2.0;  
 public const double Z = 3.0;  
}

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. In the example

class A  
{  
 public const int X = B.Z + 1;  
 public const int Y = 10;  
}

class B  
{  
 public const int Z = A.Y + 1;  
}

the compiler first evaluates A.Y, then evaluates B.Z, and finally evaluates A.X, producing the values 10, 11, and 12. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if A and B were declared in separate programs, it would be possible for A.X to depend on B.Z, but B.Z could then not simultaneously depend on A.Y.

## Fields

A field is a member that represents a variable associated with an object or class. A field-declaration introduces one or more fields of a given type.

field-declaration:  
attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:  
field-modifier  
field-modifiers field-modifier

field-modifier:  
new  
public  
protected  
internal  
private  
static  
readonly  
volatile

variable-declarators:  
variable-declarator  
variable-declarators , variable-declarator

variable-declarator:  
identifier  
identifier = variable-initializer

variable-initializer:  
expression  
array-initializer

A field-declaration may include a set of attributes (§17), a new modifier (§10.3.4), a valid combination of the four access modifiers (§10.3.5), and a static modifier (§10.5.1). In addition, a field-declaration may include a readonly modifier (§10.5.2) or a volatile modifier (§10.5.3) but not both. The attributes and modifiers apply to all of the members declared by the field-declaration. It is an error for the same modifier to appear multiple times in a field declaration.

The type of a field-declaration specifies the type of the members introduced by the declaration. The type is followed by a list of variable-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 (§10.5.5) that gives the initial value of that member.

The type of a field must be at least as accessible as the field itself (§3.5.4).

The value of a field is obtained in an expression using a simple-name (§7.6.2) or a member-access (§7.6.4). The value of a non-readonly field is modified using an assignment (§7.17). The value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators (§7.6.9) and prefix increment and decrement operators (§7.7.5).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example

class A  
{  
 public static int X = 1, Y, Z = 100;  
}

is equivalent to

class A  
{  
 public static int X = 1;  
 public static int Y;  
 public static int Z = 100;  
}

## Static and instance fields

When a field declaration includes a static modifier, the fields introduced by the declaration are static fields. When no static modifier is present, the fields introduced by the declaration are instance fields. Static fields and instance fields are two of the several kinds of variables (§5) supported by C#, and at times they are referred to as static variables and instance variables, respectively.

A static field is not part of a specific instance; instead, it is shared amongst all instances of a closed type (§4.4.2). No matter how many instances of a closed class type are created, there is only ever one copy of a static field for the associated application domain.

For example:

class C<V>  
{  
 static int count = 0;

public C() {  
 count++;  
 }

public static int Count {  
 get { return count; }  
 }  
}

class Application  
{  
 static void Main() {  
 C<int> x1 = new C<int>();  
 Writeln(C<int>.Count); // Prints 1

C<double> x2 = new C<double>();  
 Writeln(C<int>.Count); // Prints 1

C<int> x3 = new C<int>();  
 Writeln(C<int>.Count); // Prints 2  
 }  
}

An instance field belongs to an instance. Specifically, every instance of a class contains a separate set of all the instance fields of that class.

When a field is referenced in a member-access (§7.6.4) of the form E.M, if M is a static field, E must denote a type containing M, and if M is an instance field, E must denote an instance of a type containing M.

The differences between static and instance members are discussed further in §10.3.7.

## Readonly fields

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.

### 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. In the example

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

private byte red, green, blue;

public Color(byte r, byte g, byte b) {  
 red = r;  
 green = g;  
 blue = b;  
 }  
}

the Black, White, Red, Green, and Blue members cannot be declared as const members because their values cannot be computed at compile-time. However, declaring them static readonly instead has much the same effect.

### 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. Consider an application that consists of two separate programs:

using System;

namespace Program1  
{  
 public class Utils  
 {  
 public static readonly int X = 1;  
 }  
}

namespace Program2  
{  
 class Test  
 {  
 static void Main() {  
 Writeln(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 Writeln 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 Writeln 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.

### 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 (§8.12). 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 must be one of the following:

* A reference-type.
* The type byte, sbyte, short, ushort, int, uint, char, float, bool, System.IntPtr, or System.UIntPtr.
* An enum-type having an enum base type of byte, sbyte, short, ushort, int, or uint.

The example

using System;  
using System.Threading;

class Test  
{  
 public static int result;   
 public static volatile bool finished;

static void Thread2() {  
 result = 143;   
 finished = true;   
 }

static void Main() {  
 finished = false;

// Run Thread2() in a new thread  
 new Thread(new ThreadStart(Thread2)).Start();

// Wait for Thread2 to signal that it has a result by setting  
 // finished to true.  
 for (;;) {  
 if (finished) {  
 Writeln("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 must 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.

### Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (§5.2) of the field’s type. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never “uninitialized”. The example

using System;

class Test  
{  
 static bool b;  
 int i;

static void Main() {  
 Test t = new Test();  
 Writeln("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.

### Variable initializers

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.

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();  
 Writeln("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.

The default value initialization described in §10.5.4 occurs for all fields, including fields that have variable initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields in that instance are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state. 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() {  
 Writeln("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.

### Static field initialization

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor (§10.12) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. The example

using System;

class Test   
{   
 static void Main() {  
 Writeln("{0} {1}", B.Y, A.X);  
 }

public static int F(string s) {  
 Writeln(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() {  
 Writeln("{0} {1}", B.Y, A.X);  
 }

public static int F(string s) {  
 Writeln(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 must be:

Init B  
Init A  
1 1

because the rules for when static constructors execute (as defined in §10.12) provide that B’s static constructor (and hence B’s static field initializers) must run before A’s static constructor and field initializers.

### Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (§10.11.1) of that class. The variable initializers are executed in the textual order in which they appear in the class declaration. The class instance creation and initialization process is described further in §10.11.

A variable initializer for an instance field cannot reference the instance being created. Thus, it is a compile-time error to reference this in a variable initializer, as it is a compile-time error for a variable initializer to reference any instance member through a simple-name. In the example

class A  
{  
 int x = 1;  
 int y = x + 1; // Error, reference to instance member of this  
}

the variable initializer for y results in a compile-time error because it references a member of the instance being created.

## Methods

A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using method-declarations:

method-declaration:  
method-header method-body

method-header:  
attributesopt method-modifiersopt partialopt return-type member-name type-parameter-listopt  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

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 (§17) and a valid combination of the four access modifiers (§10.3.5), the new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6), and extern (§10.6.7) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

* The declaration includes a valid combination of access modifiers (§10.3.5).
* The declaration does not include the same modifier multiple times.
* The declaration includes at most one of the following modifiers: static, virtual, and override.
* The declaration includes at most one of the following modifiers: new and override.
* If the declaration includes the abstract modifier, then the declaration does not include any of the following modifiers: static, virtual, sealed or extern.
* If the declaration includes the private modifier, then the declaration does not include any of the following modifiers: virtual, override, or abstract.
* If the declaration includes the sealed modifier, then the declaration also includes the override modifier.
* If the declaration includes the partial modifier, then it does not include any of the following modifiers: new, public, protected, internal, private, virtual, sealed, override, abstract, or extern.

A method that has the async modifier is an async function and follows the rules described in §10.14.

The return-type of a method declaration specifies the type of the value computed and returned by the method. The return-type is void if the method does not return a value. If the declaration includes the partial modifier, then the return type must be void.

The member-name specifies the name of the method. Unless the method is an explicit interface member implementation (§13.4.1), the member-name is simply an identifier. For an explicit interface member implementation, the member-name consists of an interface-type followed by a “.” and an identifier.

The optional type-parameter-list specifies the type parameters of the method (§10.1.3). If a type-parameter-list is specified the method is a generic method. If the method has an extern modifier, a type-parameter-list cannot be specified.

The optional formal-parameter-list specifies the parameters of the method (§10.6.1).

The optional type-parameter-constraints-clauses specify constraints on individual type parameters (§10.1.5) and may only be specified if a type-parameter-list is also supplied, and the method does not have an override modifier.

The return-type and each of the types referenced in the formal-parameter-list of a method must be at least as accessible as the method itself (§3.5.4).

For abstract and extern methods, the method-body consists simply of a semicolon. For partial methods the method-body may consist of either a semicolon or a block. For all other methods, the method-body consists of a block, which specifies the statements to execute when the method is invoked.

If the method-body consists of a semicolon, the the declaration may not include the async modifier.

The name, the type parameter list and the formal parameter list of a method define the signature (§3.6) of the method. Specifically, the signature of a method consists of its name, the number of type parameters and the number, modifiers, and types of its formal parameters. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method.The return type is not part of a method’s signature, nor are the names of the type parameters or the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by ref and out.

The method’s type-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 must have different names.

## Method parameters

The parameters of a method, if any, are declared by the method’s formal-parameter-list.

formal-parameter-list:  
fixed-parameters  
fixed-parameters , parameter-array  
parameter-array

fixed-parameters:  
fixed-parameter  
fixed-parameters , fixed-parameter

fixed-parameter:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

parameter-modifier:  
ref  
out  
this

parameter-array:  
attributesopt params array-type identifier

The formal parameter list consists of one or more comma-separated parameters of which only the last may be a parameter-array.

A fixed-parameter consists of an optional set of attributes (§17), an optional ref, out or this modifier, a type, an identifier and an optional default-argument. Each fixed-parameter declares a parameter of the given type with the given name. The this modifier designates the method as an extension method and is only allowed on the first parameter of a static method. Extension methods are further described in §10.6.9.

A fixed-parameter with a default-argument is known as an optional parameter, whereas a fixed-parameter without a default-argument is a required parameter. A required parameter may not appear after an optional parameter in a formal-parameter-list.

A ref or out parameter cannot have a default-argument. The expression in a default-argument must be one of the following:

* a constant-expression
* an expression of the form new S() where S is a value type
* an expression of the form default(S) where S is a value type

The expression must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

If optional parameters occur in an implementing partial method declaration (§10.2.7) , an explicit interface member implementation (§13.4.1) or in a single-parameter indexer declaration (§10.9) the compiler should give a warning, since these members can never be invoked in a way that permits arguments to be omitted.

A parameter-array consists of an optional set of attributes (§17), a params modifier, an array-type, and an identifier. A parameter array declares a single parameter of the given array type with the given name. The array-type of a parameter array must be a single-dimensional array type (§12.1). In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. Parameter arrays are described further in §10.6.1.4.

A parameter-array may occur after an optional parameter, but cannot have a default value – the omission of arguments for a parameter-array would instead result in the creation of an empty array.

The following example illustrates different kinds of parameters:

public void M(  
 ref int i,  
 decimal d,  
 bool b = false,  
 bool? n = false,  
 string s = "Hello",  
 object o = null,  
 T t = default(T),  
 params int[] a  
) { }

In the formal-parameter-list for M, i is a required ref parameter, d is a required value parameter, b, s, o and t are optional value parameters and a is a parameter array.

A method declaration creates a separate declaration space for parameters, type parameters and local variables. Names are introduced into this declaration space by the type parameter list and the formal parameter list of the method and by local variable declarations in the block of the method. It is an error for two members of a method declaration space to have the same name. It is an error for the method declaration space and the local variable declaration space of a nested declaration space to contain elements with the same name.

A method invocation (§7.6.5.1) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the block of a method, formal parameters can be referenced by their identifiers in simple-name expressions (§7.6.2).

There are four kinds of formal parameters:

* Value parameters, which are declared without any modifiers.
* Reference parameters, which are declared with the ref modifier.
* Output parameters, which are declared with the out modifier.
* Parameter arrays, which are declared with the params modifier.

As described in §3.6, the ref and out modifiers are part of a method’s signature, but the params modifier is not.

### Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression that is implicitly convertible (§6.1) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter—they have no effect on the actual argument given in the method invocation.

### Reference parameters

A parameter declared with a ref modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword ref followed by a variable-reference (§5.3.3) of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

A method declared as an iterator (§10.14) cannot have reference parameters.

The example

using System;

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

static void Main() {  
 int i = 1, j = 2;  
 Swap(ref i, ref j);  
 Writeln("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.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the example

class A  
{  
 string s;

void F(ref string a, ref string b) {  
 s = "One";  
 a = "Two";  
 b = "Three";  
 }

void G() {  
 F(ref s, ref s);  
 }  
}

the invocation of F in G passes a reference to s for both a and b. Thus, for that invocation, the names s, a, and b all refer to the same storage location, and the three assignments all modify the instance field s.

### Output parameters

A parameter declared with an out modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword out followed by a variable-reference (§5.3.3) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

A method declared as a partial method (§10.2.7) or an iterator (§10.14) cannot have output parameters.

Output parameters are typically used in methods that produce multiple return values. For example:

using System;

class Test  
{  
 static void SplitPath(string path, out string dir, out string name) {  
 int i = path.Length;  
 while (i > 0) {  
 char ch = path[i – 1];  
 if (ch == '\\' || ch == '/' || ch == ':') break;  
 i--;  
 }  
 dir = path.Substring(0, i);  
 name = path.Substring(i);  
 }

static void Main() {  
 string dir, name;  
 SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);  
 Writeln(dir);  
 Writeln(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.

### Parameter arrays

A parameter declared with a params modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types string[] and string[][] can be used as the type of a parameter array, but the type string[,] can not. It is not possible to combine the params modifier with the modifiers ref and out.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

* The argument given for a parameter array can be a single expression that is implicitly convertible (§6.1) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
* Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression that is implicitly convertible (§6.1) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter (§10.6.1.1) of the same type.

The example

using System;

class Test  
{  
 static void F(params int[] args) {  
 Console.Write("Array contains {0} elements:", args.Length);  
 foreach (int i in args)   
 Console.Write(" {0}", i);  
 Writeln();  
 }

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 a as a value parameter. The second invocation of F automatically creates a four-element int[] with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of F creates a zero-element int[] and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

F(new int[] {10, 20, 30, 40});  
F(new int[] {});

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form (§7.5.3.1). The expanded form of a method is available only if the normal form of the method is not applicable and only if an applicable method with the same signature as the expanded form is not already declared in the same type.

The example

using System;

class Test  
{  
 static void F(params object[] a) {  
 Writeln("F(object[])");  
 }

static void F() {  
 Writeln("F()");  
 }

static void F(object a0, object a1) {  
 Writeln("F(object,object)");  
 }

static void Main() {  
 F();  
 F(1);  
 F(1, 2);  
 F(1, 2, 3);  
 F(1, 2, 3, 4);  
 }  
}

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.

When the type of a parameter array is object[], a potential ambiguity arises between the normal form of the method and the expended form for a single object parameter. The reason for the ambiguity is that an object[] is itself implicitly convertible to type object. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

The example

using System;

class Test  
{  
 static void F(params object[] args) {  
 foreach (object o in args) {  
 Console.Write(o.GetType().FullName);  
 Console.Write(" ");  
 }  
 Writeln();  
 }

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[]).

## Static and instance methods

When a method declaration includes a static modifier, that method is said to be a static method. When no static modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to this in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as this (§7.6.7).

When a method is referenced in a member-access (§7.6.4) of the form E.M, if M is a static method, E must denote a type containing M, and if M is an instance method, E must denote an instance of a type containing M.

The differences between static and instance members are discussed further in §10.3.7.

## 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 (§10.6.4).

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:

* First, overload resolution is applied to C, N, and A, to select a specific method M from the set of methods declared in and inherited by C. This is described in §7.6.5.1.
* Then, if M is a 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.
* Otherwise, if R contains an override of M, then this is the most derived implementation of M.
* Otherwise, the most derived implementation of M with respect to R is the same as the most derived implementation of M with respect to the direct super class of R.

The following example illustrates the differences between virtual and non-virtual methods:

using System;

class A  
{  
 public void F() { Writeln("A.F"); }

public virtual void G() { Writeln("A.G"); }  
}

class B: A  
{  
 new public void F() { Writeln("B.F"); }

public override void G() { Writeln("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.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. In the example

using System;

class A  
{  
 public virtual void F() { Writeln("A.F"); }  
}

class B: A  
{  
 public override void F() { Writeln("B.F"); }  
}

class C: B  
{  
 new public virtual void F() { Writeln("C.F"); }  
}

class D: C  
{  
 public override void F() { Writeln("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.

## 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 super class type of C, starting with the direct super class type of C and continuing with each successive direct super class type, until in a given super 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, or if it is internal and declared in the same program as C.

A compile-time error occurs unless all of the following are true for an override declaration:

* An overridden base method can be located as described above.
* There is exactly one such overridden base method. This restriction has effect only if the super class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
* The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
* The overridden base method is not a sealed method.
* The override method and the overridden base method have the same return type.
* The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method. However, if the overridden base method is protected internal and it is declared in a different assembly than the assembly containing the override method then the override method’s declared accessibility must be protected.
* The override declaration does not specify type-parameter-constraints-clauses. Instead the constraints are inherited from the overridden base method. Note that constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not legal when explicitly specified, such as value types or sealed types.

The following example demonstrates how the overriding rules work for generic classes:

abstract class C<T>  
{  
 public virtual T F() {...}

public virtual C<T> G() {...}

public virtual void H(C<T> x) {...}  
}

class D: C<string>  
{  
 public override string F() {...} // 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) {...} // Error, should be C<U>  
}

An override declaration can access the overridden base method using a base-access (§7.6.8). In the example

class A  
{  
 int x;

public virtual void PrintFields() {  
 Writeln("x = {0}", x);  
 }  
}

class B: A  
{  
 int y;

public override void PrintFields() {  
 base.PrintFields();  
 Writeln("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.

Only by including an override modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. In the example

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 public virtual void F() {} // Warning, hiding inherited F()  
}

the F method in B does not include an override modifier and therefore does not override the F method in A. Rather, the F method in B hides the method in A, and a warning is reported because the declaration does not include a new modifier.

In the example

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 new private void F() {} // Hides A.F within body of B  
}

class C: B  
{  
 public override void F() {} // 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.

## Sealed methods

When an instance method declaration includes a sealed modifier, that method is said to be a sealed method. If an instance method declaration includes the sealed modifier, it must also include the override modifier. Use of the sealed modifier prevents a derived class from further overriding the method.

The example

using System;

class A  
{  
 public virtual void F() {  
 Writeln("A.F");  
 }

public virtual void G() {  
 Writeln("A.G");  
 }  
}

class B: A  
{  
 sealed override public void F() {  
 Writeln("B.F");  
 }

override public void G() {  
 Writeln("B.G");  
 }   
}

class C: B  
{  
 override public void G() {  
 Writeln("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.

## 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 (§10.1.1.1).

In the example

public abstract class Shape  
{  
 public abstract void Paint(Graphics g, Rectangle r);  
}

public class Ellipse: Shape  
{  
 public override void Paint(Graphics g, Rectangle r) {  
 g.DrawEllipse(r);  
 }  
}

public class Box: Shape  
{  
 public override void Paint(Graphics g, Rectangle r) {  
 g.DrawRect(r);  
 }  
}

the Shape class defines the abstract notion of a geometrical shape object that can paint itself. The Paint method is abstract because there is no meaningful default implementation. The Ellipse and Box classes are concrete Shape implementations. Because these classes are non-abstract, they are required to override the Paint method and provide an actual implementation.

It is a compile-time error for a base-access (§7.6.8) to reference an abstract method. In the example

abstract class A  
{  
 public abstract void F();  
}

class B: A  
{  
 public override void F() {  
 base.F(); // Error, base.F is abstract  
 }  
}

a compile-time error is reported for the base.F() invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

using System;

class A  
{  
 public virtual void F() {  
 Writeln("A.F");  
 }  
}

abstract class B: A  
{  
 public abstract override void F();  
}

class C: B  
{  
 public override void F() {  
 Writeln("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.

## External methods

When a method declaration includes an extern modifier, that method is said to be an external method. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the method-body of an external method simply consists of a semicolon. An external method may not be generic.

The extern modifier is typically used in conjunction with a DllImport attribute (§17.5.1), allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a DllImport attribute, the method declaration must also include a static modifier. This example demonstrates the use of the extern modifier and the DllImport attribute:

using System.Text;  
using System.Security.Permissions;  
using System.Runtime.InteropServices;

class Path  
{  
 [DllImport("kernel32", SetLastError=true)]  
 static extern bool CreateDirectory(string name, SecurityAttribute sa);

[DllImport("kernel32", SetLastError=true)]  
 static extern bool RemoveDirectory(string name);

[DllImport("kernel32", SetLastError=true)]  
 static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

[DllImport("kernel32", SetLastError=true)]  
 static extern bool SetCurrentDirectory(string name);  
}

## Partial methods

When a method declaration includes a partial modifier, that method is said to be a partial method. Partial methods can only be declared as members of partial types (§10.2), and are subject to a number of restrictions. Partial methods are further described in §10.2.7.

## Extension methods

When the first parameter of a method includes the this modifier, that method is said to be an extension method. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than this, and the parameter type cannot be a pointer type.

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;  
 }  
}

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax (§7.6.5.2), using the receiver expression as the first argument.

The following program uses the extension methods declared above:

static class Program  
{  
 static void Main() {  
 string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in strings.Slice(1, 2)) {  
 Writeln(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)) {  
 Writeln(Extensions.ToInt32(s));  
 }  
 }  
}

## Method body

The method-body of a method declaration consists of either a block or a semicolon.

Abstract and external method declarations do not provide a method implementation, so their method bodies simply consist of a semicolon. For any other method, the method body is a block (§8.2) that contains the statements to execute when that method is invoked.

The result 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 result type of a non-async method is its return type, and the result type of an async method with return type System.Threading.Tasks.Task<*T*> is *T*.

When the result type of a method is void, return statements (§8.9.4) in that method’s body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its current caller.

When the result type of a method is not void, each return statement in that method’s body must specify an expression that is implicitly convertible to the result type. The endpoint of the method body of a value-returning method must not be reachable. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.

In the example

class A  
{  
 public int F() {} // Error, return value required

public int G() {  
 return 1;  
 }

public int H(bool b) {  
 if (b) {  
 return 1;  
 }  
 else {  
 return 0;  
 }  
 }  
}

the value-returning F method results in a compile-time error because control can flow off the end of the method body. The G and H methods are correct because all possible execution paths end in a return statement that specifies a return value.

## Method overloading

The method overload resolution rules are described in §7.5.2.

## Properties

A property is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have accessors that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object’s attributes; furthermore, they permit such attributes to be computed.

Properties are declared using property-declarations:

property-declaration:  
attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers:  
property-modifier  
property-modifiers property-modifier

property-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

member-name:  
identifier  
interface-type . identifier

A property-declaration may include a set of attributes (§17) and a valid combination of the four access modifiers (§10.3.5), the new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6), and extern (§10.6.7) modifiers.

Property declarations are subject to the same rules as method declarations (§10.6) with regard to valid combinations of modifiers.

The type of a property declaration specifies the type of the property introduced by the declaration, and the member-name specifies the name of the property. Unless the property is an explicit interface member implementation, the member-name is simply an identifier. For an explicit interface member implementation (§13.4.1), the member-name consists of an interface-type followed by a “.” and an identifier.

The type of a property must be at least as accessible as the property itself (§3.5.4).

The accessor-declarations, which must be enclosed in “{” and “}” tokens, declare the accessors (§10.7.2) of the property. The accessors specify the executable statements associated with reading and writing the property.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a ref or out argument.

When a property declaration includes an extern modifier, the property is said to be an external property. Because an external property declaration provides no actual implementation, each of its accessor-declarations consists of a semicolon.

## Static and instance properties

When a property declaration includes a static modifier, the property is said to be a static property. When no static modifier is present, the property is said to be an instance property.

A static property is not associated with a specific instance, and it is a compile-time error to refer to this in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as this (§7.6.7) in the accessors of that property.

When a property is referenced in a member-access (§7.6.4) of the form E.M, if M is a static property, E must denote a type containing M, and if M is an instance property, E must denote an instance of a type containing M.

The differences between static and instance members are discussed further in §10.3.7.

## 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-declarationopt  
set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:  
attributesopt accessor-modifieropt  get accessor-body

set-accessor-declaration:  
attributesopt accessor-modifieropt set accessor-body

accessor-modifier:  
protected  
internal  
private  
protected internal  
internal protected

accessor-body:  
block  
;

The accessor declarations consist of a get-accessor-declaration, a set-accessor-declaration, or both. Each accessor declaration consists of the token get or set followed by an optional accessor-modifier and an accessor-body.

The use of accessor-modifiers is governed by the following restrictions:

* An accessor-modifier may not be used in an interface or in an explicit interface member implementation.
* For a property or indexer that has no override modifer, 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 modifer, an accessor must match the accessor-modifier, if any, of the accessor being overridden.
* The accessor-modifier must declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
* If the property or indexer has a declared accessibility of public, the accessor-modifier may be either protected internal, internal, protected, or private.
* If the property or indexer has a declared accessibility of protected internal, the accessor-modifier may be either internal, protected, or private.
* If the property or indexer has a declared accessibility of internal or protected, the accessor-modifier must be private.
* If the property or indexer has a declared accessibility of private, no accessor-modifier may be used.

For abstract and extern properties, the accessor-body for each accessor specified is simply a semicolon. A non-abstract, non-extern property may be an automatically implemented property, in which case both get and set accessors must be given, both with a semicolon body (§10.7.3). For the accessors of any other non-abstract, non-extern property, the accessor-body is a block which specifies the statements to be executed when the corresponding accessor is invoked.

A get accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the get accessor of the property is invoked to compute the value of the property (§7.1.1). The body of a get accessor must conform to the rules for value-returning methods described in §10.6.10. In particular, all return statements in the body of a get accessor must specify an expression that is implicitly convertible to the property type. Furthermore, the endpoint of a get accessor must not be reachable.

A set accessor corresponds to a method with a single value parameter of the property type and a void return type. The implicit parameter of a set accessor is always named value. When a property is referenced as the target of an assignment (§7.17), or as the operand of ++ or -- (§7.6.9, §7.7.5), the set accessor is invoked with an argument (whose value is that of the right-hand side of the assignment or the operand of the ++ or -- operator) that provides the new value (§7.17.1). The body of a set accessor must conform to the rules for void methods described in §10.6.10. In particular, return statements in the set accessor body are not permitted to specify an expression. Since a set accessor implicitly has a parameter named value, it is a compile-time error for a local variable or constant declaration in a set accessor to have that name.

Based on the presence or absence of the get and set accessors, a property is classified as follows:

* A property that includes both a get accessor and a set accessor is said to be a read-write property.
* A property that has only a get accessor is said to be a read-only property. It is a compile-time error for a read-only property to be the target of an assignment.
* A property that has only a set accessor is said to be a write-only property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

public class Button: Control  
{  
 private string caption;

public string Caption {  
 get {  
 return caption;  
 }  
 set {  
 if (caption != value) {  
 caption = value;  
 Repaint();  
 }  
 }  
 }

public override void Paint(Graphics g, Rectangle r) {  
 // Painting code goes here  
 }  
}

the Button control declares a public Caption property. The get accessor of the Caption property returns the string stored in the private caption field. The set accessor checks 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 fully update the state of the object.

Given the Button class above, the following is an example of use of the Caption property:

Button okButton = new Button();  
okButton.Caption = "OK"; // Invokes set accessor  
string s = okButton.Caption; // Invokes get accessor

Here, the set accessor is invoked by assigning a value to the property, and the get accessor is invoked by referencing the property in an expression.

The get and set accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. As such, it is not possible for the two accessors of a read-write property to have different accessibility. The example

class A  
{  
 private string name;

public string Name { // Error, duplicate member name  
 get { return name; }  
 }

public string Name { // Error, duplicate member name  
 set { name = value; }  
 }  
}

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. In the example

class A  
{  
 public int P {  
 set {...}  
 }  
}

class B: A  
{  
 new public int P {  
 get {...}  
 }  
}

the P property in B hides the P property in A with respect to both reading and writing. Thus, in the statements

B b = new B();  
b.P = 1; // Error, B.P is read-only  
((A)b).P = 1; // 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.

Unlike public fields, properties provide a separation between an object’s internal state and its public interface. Consider the example:

class Label  
{  
 private int x, y;  
 private string caption;

public Label(int x, int y, string caption) {  
 this.x = x;  
 this.y = y;  
 this.caption = caption;  
 }

public int X {  
 get { return x; }  
 }

public int Y {  
 get { return y; }  
 }

public Point Location {  
 get { return new Point(x, y); }  
 }

public string Caption {  
 get { return caption; }  
 }  
}

Here, the Label class uses two int fields, x and y, to store its location. The location is publicly exposed both as an X and a Y property and as a Location property of type Point. If, in a future version of Label, it becomes more convenient to store the location as a Point internally, the change can be made without affecting the public interface of the class:

class Label  
{  
 private Point location;  
 private string caption;

public Label(int x, int y, string caption) {  
 this.location = new Point(x, y);  
 this.caption = caption;  
 }

public int X {  
 get { return location.x; }  
 }

public int Y {  
 get { return location.y; }  
 }

public Point Location {  
 get { return location; }  
 }

public string Caption {  
 get { return caption; }  
 }  
}

Had x and y instead been public readonly fields, it would have been impossible to make such a change to the Label class.

Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as inlining, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a get accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for get accessors to have observable side-effects. In the example

class Counter  
{  
 private int next;

public int Next {  
 get { return next++; }  
 }  
}

the value of the Next property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side-effect, and the property should be implemented as a method instead.

The “no side-effects” convention for get accessors doesn’t mean that get accessors should always be written to simply return values stored in fields. Indeed, get accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed get accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

using System.IO;

public class Console  
{  
 private static TextReader reader;  
 private static TextWriter writer;  
 private static TextWriter error;

public static TextReader In {  
 get {  
 if (reader == null) {  
 reader = new StreamReader(Console.OpenStandardInput());  
 }  
 return reader;  
 }  
 }

public static TextWriter Out {  
 get {  
 if (writer == null) {  
 writer = new StreamWriter(Console.OpenStandardOutput());  
 }  
 return writer;  
 }  
 }

public static TextWriter Error {  
 get {  
 if (error == null) {  
 error = new StreamWriter(Console.OpenStandardError());  
 }  
 return error;  
 }  
 }  
}

The Console class contains three properties, In, Out, and Error, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the Console class can delay their initialization until they are actually used. For example, upon first referencing the Out property, as in

Console.Out.WriteLine("hello, world");

the underlying TextWriter for the output device is created. But if the application makes no reference to the In and Error properties, then no objects are created for those devices.

## 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 following example:

public class Point {  
 public int X { get; set; } // automatically implemented  
 public int Y { get; set; } // automatically implemented  
}

is equivalent to the following declaration:

public class Point {  
 private int x;  
 private int y;  
 public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Because the backing field is inaccessible, it can be read and written only through the property accessors, even within the containing type. This means that automatically implemented read-only or write-only properties do not make sense, and 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; }  
}

This restriction also means that definite assignment of struct types with auto-implemented properties can only be achieved using the standard constructor of the struct, since assigning to the property itself requires the struct to be definitely assigned. This means that user-defined constructors must call the default constructor.

## Accessibility

If an accessor has an accessor-modifier, the accessibility domain (§3.5.2) of the accessor is determined using the declared accessibility of the accessor-modifier. If an accessor does not have an accessor-modifier, the accessibility domain of the accessor is determined from the declared accessibility of the property or indexer.

The presence of an accessor-modifier never affects member lookup (§7.3) or overload resolution (§7.5.3). The modifiers on the property or indexer always determine which property or indexer is bound to, regardless of the context of the access.

Once a particular property or indexer has been selected, the accessibility domains of the specific accessors involved are used to determine if that usage is valid:

* If the usage is as a value (§7.1.1), the get accessor must exist and be accessible.
* If the usage is as the target of a simple assignment (§7.17.1), the set accessor must exist and be accessible.
* If the usage is as the target of compound assignment (§7.17.2), or as the target of the ++ or -- operators (§7.5.9, §7.6.5), both the get accessors and the set accessor must exist and be accessible.

In the following example, the property A.Text is hidden by the property B.Text, even in contexts where only the set accessor is called. In contrast, the property B.Count is not accessible to class M, so the accessible property A.Count is used instead.

class A  
{  
 public string Text {  
 get { return "hello"; }  
 set { }  
 }

public int Count {  
 get { return 5; }  
 set { }  
 }  
}

class B: A  
{  
 private string text = "goodbye";   
 private int count = 0;

new public string Text {  
 get { return text; }  
 protected set { text = value; }  
 }

new protected int Count {   
 get { return count; }  
 set { count = value; }  
 }  
}

class M  
{  
 static void Main() {  
 B b = new B();  
 b.Count = 12; // Calls A.Count set accessor  
 int i = b.Count; // Calls A.Count get accessor  
 b.Text = "howdy"; // Error, B.Text set accessor not accessible  
 string s = b.Text; // Calls B.Text get accessor  
 }  
}

An accessor that is used to implement an interface may not have an accessor-modifier. If only one accessor is used to implement an interface, the other accessor may be declared with an accessor-modifier:

public interface I  
{  
 string Prop { get; }  
}

public class C: I  
{  
 public Prop {  
 get { return "April"; } // Must not have a modifier here  
 internal set {...} // Ok, because I.Prop has no set accessor  
 }  
}

## Virtual, sealed, override, and abstract accessors

A virtual property declaration specifies that the accessors of the property are virtual. The virtual modifier applies to both accessors of a read-write property—it is not possible for only one accessor of a read-write property to be virtual.

An abstract property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an abstract property declaration provides no actual implementation, its accessor-body simply consists of a semicolon.

A property declaration that includes both the abstract and override modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are only permitted in abstract classes (§10.1.1.1).The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an override directive. This is known as an overriding property declaration. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration must specify the exact same accessibility modifiers, type, and name as the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property must include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

An overriding property declaration may include the sealed modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in §10.6.3, §10.6.4, §10.6.5, and §10.6.6 apply as if accessors were methods of a corresponding form:

* A get accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
* A set accessor corresponds to a method with a single value parameter of the property type, a void return type, and the same modifiers as the containing property.

In the example

abstract class A  
{  
 int y;

public virtual int X {  
 get { return 0; }  
 }

public virtual int Y {  
 get { return y; }  
 set { y = value; }  
 }

public abstract int Z { get; set; }  
}

X is a virtual read-only property, Y is a virtual read-write property, and Z is an abstract read-write property. Because Z is abstract, the containing class A must also be declared abstract.

A class that derives from A is show below:

class B: A  
{  
 int z;

public override int X {  
 get { return base.X + 1; }  
 }

public override int Y {  
 set { base.Y = value < 0? 0: value; }  
 }

public override int Z {  
 get { return z; }  
 set { z = value; }  
 }  
}

Here, the declarations of X, Y, and Z are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The get accessor of X and the set accessor of Y use the base keyword to access the inherited accessors. The declaration of Z overrides both abstract accessors—thus, there are no outstanding abstract function members in B, and B is permitted to be a non-abstract class.

When a property is declared as an override, any overridden accessors must be accessible to the overriding code. In addition, the declared accessibility of both the property or indexer itself, and of the accessors, must match that of the overridden member and accessors. For example:

public class B  
{  
 public virtual int P {  
 protected set {...}  
 get {...}  
 }  
}

public class D: B  
{  
 public override int P {  
 protected set {...} // Must specify protected here  
 get {...} // Must not have a modifier here  
 }  
}

## Events

An event is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying event handlers.

Events are declared using event-declarations:

event-declaration:  
attributesopt event-modifiersopt event type variable-declarators ;  
attributesopt event-modifiersopt 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:  
attributesopt remove block

An event-declaration may include a set of attributes (§17) and a valid combination of the four access modifiers (§10.3.5), the new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6), and extern (§10.6.7) modifiers.

Event declarations are subject to the same rules as method declarations (§10.6) with regard to valid combinations of modifiers.

The type of an event declaration must be a delegate-type (§4.2), and that delegate-type must be at least as accessible as the event itself (§3.5.4).

An event declaration may include event-accessor-declarations. However, if it does not, for non-extern, non-abstract events, the compiler supplies them automatically (§10.8.1); for extern events, the accessors are provided externally.

An event declaration that omits event-accessor-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 brace-delimited 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 (§7.17.3). These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since += and -= are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

In an operation of the form x += y or x -= y, when x is an event and the reference takes place outside the type that contains the declaration of x, the result of the operation has type void (as opposed to having the type of x, with the value of x after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the Button class:

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control  
{  
 public event EventHandler Click;  
}

public class LoginDialog: Form  
{  
 Button OkButton;  
 Button CancelButton;

public LoginDialog() {  
 OkButton = new Button(...);  
 OkButton.Click += new EventHandler(OkButtonClick);  
 CancelButton = new Button(...);  
 CancelButton.Click += new EventHandler(CancelButtonClick);  
 }

void OkButtonClick(object sender, EventArgs e) {  
 // Handle OkButton.Click event  
 }

void CancelButtonClick(object sender, EventArgs e) {  
 // Handle CancelButton.Click event  
 }  
}

Here, the LoginDialog instance constructor creates two Button instances and attaches event handlers to the Click events.

## Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be abstract or extern, and must not explicitly include event-accessor-declarations. Such an event can be used in any context that permits a field. The field contains a delegate (§15) which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains null.

In the example

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control  
{  
 public event EventHandler Click;

protected void OnClick(EventArgs e) {  
 if (Click != null) Click(this, e);  
 }

public void Reset() {  
 Click = null;  
 }  
}

Click is used as a field within the Button class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The OnClick method in the Button class “raises” the Click event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the Button class, the Click member can 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.

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock (§8.12) on the containing object for an instance event, or the type object (§7.6.10.6) for a static event.

Thus, an instance event declaration of the form:

class X  
{  
 public event D Ev;  
}

will be compiled to something equivalent to:

class X  
{  
 private D \_\_Ev; // field to hold the delegate

public event D Ev {  
 add {  
 /\* add the delegate in a thread safe way \*/  
 }

remove {  
 /\* remove the delegate in a thread safe way \*/  
 }  
 }  
}

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 (§7.6.4). The name “\_\_Ev” is arbitrary; the hidden field could have any name or no name at all.

## Event accessors

Event declarations typically omit event-accessor-declarations, as in the Button example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include event-accessor-declarations and use a private mechanism for storing the list of event handlers.

The event-accessor-declarations of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an add-accessor-declaration and a remove-accessor-declaration. Each accessor declaration consists of the token add or remove followed by a block. The block associated with an add-accessor-declaration specifies the statements to execute when an event handler is added, and the block associated with a remove-accessor-declaration specifies the statements to execute when an event handler is removed.

Each add-accessor-declaration and remove-accessor-declaration corresponds to a method with a single value parameter of the event type and a void return type. The implicit parameter of an event accessor is named value. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is += then the add accessor is used, 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 must conform to the rules for void methods described in §10.6.10. In particular, return statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named value, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

In the example

class Control: Component  
{  
 // Unique keys for events  
 static readonly object mouseDownEventKey = new object();  
 static readonly object mouseUpEventKey = new object();

// Return event handler associated with key  
 protected Delegate GetEventHandler(object key) {...}

// Add event handler associated with key  
 protected void AddEventHandler(object key, Delegate handler) {...}

// 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.

## Static and instance events

When an event declaration includes a static modifier, the event is said to be a static event. When no static modifier is present, the event is said to be an instance event.

A static event is not associated with a specific instance, and it is a compile-time error to refer to this in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as this (§7.6.7) in the accessors of that event.

When an event is referenced in a member-access (§7.6.4) of the form E.M, if M is a static event, E must denote a type containing M, and if M is an instance event, E must denote an instance of a type containing M.

The differences between static and instance members are discussed further in §10.3.7.

## Virtual, sealed, override, and abstract accessors

A virtual event declaration specifies that the accessors of that event are virtual. The virtual modifier applies to both accessors of an event.

An abstract event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an abstract event declaration provides no actual implementation, it cannot provide brace-delimited event-accessor-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 (§10.1.1.1).

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an override modifier. This is known as an overriding event declaration. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the overridden event.

An overriding event declaration may include the sealed modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed.

It is a compile-time error for an overriding event declaration to include a new modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in §10.6.3, §10.6.4, §10.6.5, and §10.6.6 apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a void return type, and the same modifiers as the containing event.

## 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:  
attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers:  
indexer-modifier  
indexer-modifiers indexer-modifier

indexer-modifier:  
new  
public  
protected  
internal  
private   
virtual  
sealed  
override  
abstract  
extern

indexer-declarator:  
type this [ formal-parameter-list ]  
type interface-type . this [ formal-parameter-list ]

An indexer-declaration may include a set of attributes (§17) and a valid combination of the four access modifiers (§10.3.5), the new (§10.3.4), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6), and extern (§10.6.7) modifiers.

Indexer declarations are subject to the same rules as method declarations (§10.6) with regard to valid combinations of modifiers, with the one exception being that the static modifier is not permitted on an indexer declaration.

The modifiers virtual, override, and abstract are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract indexer can override a virtual one.

The type of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the type is followed by the keyword this. For an explicit interface member implementation, the type is followed by an interface-type, a “.”, and the keyword this. Unlike other members, indexers do not have user-defined names.

The formal-parameter-list specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method (§10.6.1), except that at least one parameter must be specified, and that the ref and out parameter modifiers are not permitted.

The type of an indexer and each of the types referenced in the formal-parameter-list must be at least as accessible as the indexer itself (§3.5.4).

The accessor-declarations (§10.7.2), which must be enclosed in “{” and “}” tokens, declare the accessors of the indexer. The accessors specify the executable statements associated with reading and writing indexer elements.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer element is not classified as a variable. Thus, it is not possible to pass an indexer element as a ref or out argument.

The formal parameter list of an indexer defines the signature (§3.6) of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer’s signature.

The signature of an indexer must differ from the signatures of all other indexers declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

* A property is identified by its name, whereas an indexer is identified by its signature.
* A property is accessed through a simple-name (§7.6.2) or a member-access (§7.6.4), whereas an indexer element is accessed through an element-access (§7.6.6.2).
* A property can be a static member, whereas an indexer is always an instance member.
* A get accessor of a property corresponds to a method with no parameters, whereas a get accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
* A set accessor of a property corresponds to a method with a single parameter named value, whereas a set accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named value.
* It is a compile-time error for an indexer accessor to declare a local variable with the same name as an indexer parameter.
* In an overriding property declaration, the inherited property is accessed using the syntax base.P, where P is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax base[E], where E is a comma separated list of expressions.

Aside from these differences, all rules defined in §10.7.2 and §10.7.3 apply to indexer accessors as well as to property accessors.

When an indexer declaration includes an extern modifier, the indexer is said to be an external indexer. Because an external indexer declaration provides no actual implementation, each of its accessor-declarations consists of a semicolon.

The example below declares a BitArray class that implements an indexer for accessing the individual bits in the bit array.

using System;

class BitArray  
{  
 int[] bits;  
 int length;

public BitArray(int length) {  
 if (length < 0) throw new ArgumentException();  
 bits = new int[((length - 1) >> 5) + 1];  
 this.length = length;  
 }

public int Length {  
 get { return length; }  
 }

public bool this[int index] {  
 get {  
 if (index < 0 || index >= length) {  
 throw new IndexOutOfRangeException();  
 }  
 return (bits[index >> 5] & 1 << index) != 0;  
 }  
 set {  
 if (index < 0 || index >= length) {  
 throw new IndexOutOfRangeException();  
 }  
 if (value) {  
 bits[index >> 5] |= 1 << index;  
 }  
 else {  
 bits[index >> 5] &= ~(1 << index);  
 }  
 }  
 }  
}

An instance of the BitArray class consumes substantially less memory than a corresponding bool[] (since each value of the former occupies only one bit instead of the latter’s one byte), but it permits the same operations as a bool[].

The following CountPrimes class uses a BitArray and the classical “sieve” algorithm to compute the number of primes between 1 and a given maximum:

class CountPrimes  
{  
 static int Count(int max) {  
 BitArray flags = new BitArray(max + 1);  
 int count = 1;  
 for (int i = 2; i <= max; i++) {  
 if (!flags[i]) {  
 for (int j = i \* 2; j <= max; j += i) flags[j] = true;  
 count++;  
 }  
 }  
 return count;  
 }

static void Main(string[] args) {  
 int max = int.Parse(args[0]);  
 int count = Count(max);  
 Writeln("Found {0} primes between 1 and {1}", count, max);  
 }  
}

Note that the syntax for accessing elements of the BitArray is precisely the same as for a bool[].

The following example shows a 26 × 10 grid class that has an indexer with two parameters. The first parameter is required to be an 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  
{  
 const int NumRows = 26;  
 const int NumCols = 10;

int[,] cells = new int[NumRows, NumCols];

public int this[char c, int col] {  
 get {  
 c = Char.ToUpper(c);  
 if (c < 'A' || c > 'Z') {  
 throw new ArgumentException();  
 }  
 if (col < 0 || col >= NumCols) {  
 throw new IndexOutOfRangeException();  
 }  
 return cells[c - 'A', col];  
 }

set {  
 c = Char.ToUpper(c);  
 if (c < 'A' || c > 'Z') {  
 throw new ArgumentException();  
 }  
 if (col < 0 || col >= NumCols) {  
 throw new IndexOutOfRangeException();  
 }  
 cells[c - 'A', col] = value;  
 }  
 }  
}

## Indexer overloading

The indexer overload resolution rules are described in §7.5.2.

## Operators

An operator is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using operator-declarations:

operator-declaration:  
attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers:  
operator-modifier  
operator-modifiers operator-modifier

operator-modifier:  
public  
static  
extern

operator-declarator:  
unary-operator-declarator  
binary-operator-declarator  
conversion-operator-declarator

unary-operator-declarator:  
type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of  
+ - ! ~ ++ -- true false

binary-operator-declarator:  
type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator:  
+  
-  
\*  
/  
%  
&  
|  
^  
<<  
right-shift  
==  
!=  
>  
<  
>=  
<=

conversion-operator-declarator:  
implicit operator type ( type identifier )  
explicit operator type ( type identifier )

operator-body:  
block  
;

There are three categories of overloadable operators: Unary operators (§10.10.1), binary operators (§10.10.2), and conversion operators (§10.10.3).

When an operator declaration includes an extern modifier, the operator is said to be an external operator. Because an external operator provides no actual implementation, its operator-body consists of a semi-colon. For all other operators, the operator-body consists of a block, which specifies the statements to execute when the operator is invoked. The block of an operator must conform to the rules for value-returning methods described in §10.6.10.

The following rules apply to all operator declarations:

* An operator declaration must include both a public and a static modifier.
* The parameter(s) of an operator must be value parameters (§5.1.4). It is a compile-time error for an operator declaration to specify ref or out parameters.
* The signature of an operator (§10.10.1, §10.10.2, §10.10.3) must differ from the signatures of all other operators declared in the same class.
* All types referenced in an operator declaration must be at least as accessible as the operator itself (§3.5.4).
* It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a super 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 super class. Thus, the new modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in §7.3.

Additional information on conversion operators can be found in §6.4.

## Unary operators

The following rules apply to unary operator declarations, where T denotes the instance type of the class or struct that contains the operator declaration:

* A unary +, -, !, or ~ operator must take a single parameter of type T or T? and can return any type.
* A unary ++ or -- operator must take a single parameter of type T or T? and must return that same type or a type derived from it.
* A unary true or false operator must take a single parameter of type T or T? and must return type bool.

The signature of a unary operator consists of the operator token (+, -, !, ~, ++, --, true, or false) and the type of the single formal parameter. The return type is not part of a unary operator’s signature, nor is the name of the formal parameter.

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 §7.12.2 and §7.20.

The following example shows an implementation and subsequent usage of operator ++ for an integer vector class:

public class IntVector  
{  
 public IntVector(int length) {...}

public int Length {...} // read-only property

public int this[int index] {...} // read-write indexer

public static IntVector operator ++(IntVector iv) {  
 IntVector temp = new IntVector(iv.Length);  
 for (int i = 0; i < iv.Length; i++)  
 temp[i] = iv[i] + 1;  
 return temp;  
 }  
}

class Test  
{  
 static void Main() {  
 IntVector iv1 = new IntVector(4); // vector of 4 x 0  
 IntVector iv2;

iv2 = iv1++; // iv2 contains 4 x 0, iv1 contains 4 x 1  
 iv2 = ++iv1; // iv2 contains 4 x 2, iv1 contains 4 x 2  
 }  
}

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (§7.6.9), and the prefix increment and decrement operators (§7.7.5). Unlike in C++, this method need not modify the value of its operand directly. In fact, modifying the operand value would violate the standard semantics of the postfix increment operator.

## 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 must take two parameters, at least one of which must have type T or T?, and can return any type.
* A binary << or >> operator must take two parameters, the first of which must have type T or T? and the second of which must have type int or int?, and can return any type.

The signature of a binary operator consists of the operator token (+, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator’s signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

* operator == and operator !=
* operator > and operator <
* operator >= and operator <=

## Conversion operators

A conversion operator declaration introduces a user-defined conversion (§6.4) which augments the pre-defined implicit and explicit conversions.

A conversion operator declaration that includes the implicit keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in §6.1.

A conversion operator declaration that includes the explicit keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions, and are described further in §6.2.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator.

For a given source type S and target type T, if S or T are nullable types, let S0 and T0 refer to their underlying types, otherwise S0 and T0 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:

* S0 and T0 are different types.
* Either S0 or T0 is the class or struct type in which the operator declaration takes place.
* Neither S0 nor T0 is an interface-type.
* Excluding user-defined conversions, a conversion does not exist from S to T or from T to S.

For the purposes of these rules, any type parameters associated with S or T are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

In the example

class C<T> {...}

class D<T>: C<T>  
{  
 public static implicit operator C<int>(D<T> value) {...} // 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, for the purposes of §10.9.3, 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 super class of D<T>.

From the second rule it follows that a conversion operator must convert either to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type C to define a conversion from C to int and from int to C, but not from int to bool.

It is not possible to directly redefine a 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. In the example

struct Convertible<T>  
{  
 public static implicit operator Convertible<T>(T value) {...}

public static explicit operator T(Convertible<T> value) {...}  
}

when type object is specified as a type argument for T, the second operator declares a conversion that already exists (an implicit, and therefore also an explicit, conversion exists from any type to type object).

In cases where a pre-defined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

* If a pre-defined implicit conversion (§6.1) exists from type S to type T, all user-defined conversions (implicit or explicit) from S to T are ignored.
* If a pre-defined explicit conversion (§6.2) exists from type S to type T, any user-defined explicit conversions from S to T are ignored. Furthermore:
* If T is an interface type, user-defined implicit conversions from S to T are ignored.
* Otherwise, user-defined implicit conversions from S to T are still considered.

For all types but object, the operators declared by the Convertible<T> type above do not conflict with pre-defined conversions. For example:

void F(int i, Convertible<int> n) {  
 i = n; // Error  
 i = (int)n; // User-defined explicit conversion  
 n = i; // User-defined implicit conversion  
 n = (Convertible<int>)i; // User-defined implicit conversion  
}

However, for type object, pre-defined conversions hide the user-defined conversions in all cases but one:

void F(object o, Convertible<object> n) {  
 o = n; // Pre-defined boxing conversion  
 o = (object)n; // Pre-defined boxing conversion  
 n = o; // User-defined implicit conversion  
 n = (Convertible<object>)o; // Pre-defined unboxing conversion  
}

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. (Note that this is the only form of member for which the return type participates in the signature.) The implicit or explicit classification of a conversion operator is not part of the operator’s signature. Thus, a class or struct cannot declare both an implicit and an explicit conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

In the example

using System;

public struct Digit  
{  
 byte value;

public Digit(byte value) {  
 if (value < 0 || value > 9) throw new ArgumentException();  
 this.value = value;  
 }

public static implicit operator byte(Digit d) {  
 return d.value;  
 }

public static explicit operator Digit(byte b) {  
 return new Digit(b);  
 }  
}

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.

## Instance constructors

An instance constructor is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using constructor-declarations:

constructor-declaration:  
attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers:  
constructor-modifier  
constructor-modifiers constructor-modifier

constructor-modifier:  
public  
protected  
internal  
private  
extern

constructor-declarator:  
identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:  
: base ( argument-listopt )  
: this ( argument-listopt )

constructor-body:  
block  
;

A constructor-declaration may include a set of attributes (§17), a valid combination of the four access modifiers (§10.3.5), and an extern (§10.6.7) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The identifier of a constructor-declarator must name the class in which the instance constructor is declared. If any other name is specified, a compile-time error occurs.

The optional formal-parameter-list of an instance constructor is subject to the same rules as the formal-parameter-list of a method (§10.6). The formal parameter list defines the signature (§3.6) of an instance constructor and governs the process whereby overload resolution (§7.5.2) selects a particular instance constructor in an invocation.

Each of the types referenced in the formal-parameter-list of an instance constructor must be at least as accessible as the constructor itself (§3.5.4).

The optional constructor-initializer specifies another instance constructor to invoke before executing the statements given in the constructor-body of this instance constructor. This is described further in §10.11.1.

When a constructor declaration includes an extern modifier, the constructor is said to be an external constructor. Because an external constructor declaration provides no actual implementation, its constructor-body consists of a semicolon. For all other constructors, the constructor-body consists of a block 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 (§10.6.10).

Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually declared in the class. If a class contains no instance constructor declarations, a default instance constructor is automatically provided (§10.11.4).

Instance constructors are invoked by object-creation-expressions (§7.6.10.1) and through constructor-initializers.

## 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-listopt) causes an instance constructor from the direct super class to be invoked. That constructor is selected using argument-list and the overload resolution rules of §7.5.3. The set of candidate instance constructors consists of all accessible instance constructors contained in the direct super class, or the default constructor (§10.11.4), if no instance constructors are declared in the direct super 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-listopt) causes an instance constructor from the class itself to be invoked. The constructor is selected using argument-list and the overload resolution rules of §7.5.3. The set of candidate instance constructors consists of all accessible instance constructors declared in the class itself. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

If an instance constructor has no constructor initializer, a constructor initializer of the form base() is implicitly provided. Thus, an instance constructor declaration of the form

C(...) {...}

is exactly equivalent to

C(...): base() {...}

The scope of the parameters given by the formal-parameter-list of an instance constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. For example:

class A  
{  
 public A(int x, int y) {}  
}

class B: A  
{  
 public B(int x, int y): base(x + y, x - y) {}  
}

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference this in an argument expression of the constructor initializer, as is it a compile-time error for an argument expression to reference any instance member through a simple-name.

## 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 super class constructor. The variable initializers are executed in the textual order in which they appear in the class declaration.

## Constructor execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed before the invocation of the super class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to that instance are executed.

Given the example

using System;

class A  
{  
 public A() {  
 PrintFields();  
 }

public virtual void PrintFields() {}

}

class B: A  
{  
 int x = 1;  
 int y;

public B() {  
 y = -1;  
 }

public override void PrintFields() {  
 Writeln("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 super 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 super 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;  
 }

public A(int n) {  
 count = n;  
 }  
}

class B: A  
{  
 double sqrt2 = Math.Sqrt(2.0);  
 ArrayList items = new ArrayList(100);  
 int max;

public B(): this(100) {  
 items.Add("default");  
 }

public B(int n): base(n – 1) {  
 max = n;  
 }  
}

contains several variable initializers; it also contains constructor initializers of both forms (base and this). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn’t valid, but merely serves to illustrate the mechanism).

using System.Collections;

class A  
{  
 int x, y, count;

public A() {  
 x = 1; // Variable initializer  
 y = -1; // Variable initializer  
 object(); // Invoke object() constructor  
 count = 0;  
 }

public A(int n) {  
 x = 1; // Variable initializer  
 y = -1; // Variable initializer  
 object(); // Invoke object() constructor  
 count = n;  
 }  
}

class B: A  
{  
 double sqrt2;  
 ArrayList items;  
 int max;

public B(): this(100) {  
 B(100); // Invoke B(int) constructor  
 items.Add("default");  
 }

public B(int n): base(n – 1) {  
 sqrt2 = Math.Sqrt(2.0); // Variable initializer  
 items = new ArrayList(100); // Variable initializer  
 A(n – 1); // Invoke A(int) constructor  
 max = n;  
 }  
}

## Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct super class. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

protected C(): base() {}

or

public C(): base() {}

where C is the name of the class. If overload resolution is unable to determine a unique best candidate for the super class constructor initializer then a compile-time error occurs.

In the example

class Message  
{  
 object sender;  
 string text;  
}

a default constructor is provided because the class contains no instance constructor declarations. Thus, the example is precisely equivalent to

class Message  
{  
 object sender;  
 string text;

public Message(): base() {}  
}

## Private constructors

When a class T declares only private instance constructors, it is not possible for classes outside the program text of T to derive from T or to directly create instances of T. Thus, if a class contains only static members and isn’t intended to be instantiated, adding an empty private instance constructor will prevent instantiation. For example:

public class Trig  
{  
 private Trig() {} // Prevent instantiation

public const double PI = 3.14159265358979323846;

public static double Sin(double x) {...}  
 public static double Cos(double x) {...}  
 public static double Tan(double x) {...}  
}

The Trig class groups related methods and constants, but is not intended to be instantiated. Therefore it declares a single empty private instance constructor. At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

## Optional instance constructor parameters

The this(...) form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters. In the example

class Text  
{  
 public Text(): this(0, 0, null) {}

public Text(int x, int y): this(x, y, null) {}

public Text(int x, int y, string s) {  
 // Actual constructor implementation  
 }  
}

the first two instance constructors merely provide the default values for the missing arguments. Both use a this(...) constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

Text t1 = new Text(); // Same as Text(0, 0, null)  
Text t2 = new Text(5, 10); // Same as Text(5, 10, null)  
Text t3 = new Text(5, 20, "Hello");

## Static constructors

A static constructor is a member that implements the actions required to initialize a closed class type. Static constructors are declared using static-constructor-declarations:

static-constructor-declaration:  
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers:  
externopt static  
static externopt

static-constructor-body:  
block  
;

A static-constructor-declaration may include a set of attributes (§17) and an extern modifier (§10.6.7).

The identifier of a static-constructor-declaration must name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

When a static constructor declaration includes an extern modifier, the static constructor is said to be an external static constructor. Because an external static constructor declaration provides no actual implementation, its static-constructor-body consists of a semicolon. For all other static constructor declarations, the static-constructor-body consists of a block 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 (§10.6.10).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a closed class type executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

* An instance of the class type is created.
* Any of the static members of the class type are referenced.

If a class contains the Main method (§3.1) in which execution begins, the static constructor for that class executes before the Main method is called.

To initialize a new closed class type, first a new set of static fields (§10.5.1) for that particular closed type is created. Each of the static fields is initialized to its default value (§5.2). Next, the static field initializers (§10.4.5.1) are executed for those static fields. Finally, the static constructor is executed.

The example

using System;

class Test  
{  
 static void Main() {  
 A.F();  
 B.F();  
 }  
}

class A  
{  
 static A() {  
 Writeln("Init A");  
 }  
 public static void F() {  
 Writeln("A.F");  
 }  
}

class B  
{  
 static B() {  
 Writeln("Init B");  
 }  
 public static void F() {  
 Writeln("B.F");  
 }  
}

must produce the output:

Init A  
A.F  
Init B  
B.F

because the execution of A’s static constructor is triggered by the call to A.F, and the execution of B’s static constructor is triggered by the call to B.F.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

The example

using System;

class A  
{  
 public static int X;

static A() {  
 X = B.Y + 1;  
 }  
}

class B  
{  
 public static int Y = A.X + 1;

static B() {}

static void Main() {  
 Writeln("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.

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 (§10.1.5). For example, the following type uses a static constructor to enforce that the type argument is an enum:

class Gen<T> where T: struct  
{  
 static Gen() {  
 if (!typeof(T).IsEnum) {  
 throw new ArgumentException("T must be an enum");  
 }  
 }  
}

## Destructors

A destructor is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a destructor-declaration:

destructor-declaration:  
attributesopt externopt ~ identifier ( ) destructor-body

destructor-body:  
block  
;

A destructor-declaration may include a set of attributes (§17).

The identifier of a destructor-declarator must name the class in which the destructor is declared. If any other name is specified, a compile-time error occurs.

When a destructor declaration includes an extern modifier, the destructor is said to be an external destructor. Because an external destructor declaration provides no actual implementation, its destructor-body consists of a semicolon. For all other destructors, the destructor-body consists of a block which specifies the statements to execute in order to destruct an instance of the class. A destructor-body corresponds exactly to the method-body of an instance method with a void return type (§10.6.10).

Destructors are not inherited. Thus, a class has no destructors other than the one which may be declared in that class.

Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in that instance’s inheritance chain are called, in order, from most derived to least derived. A destructor may be executed on any thread. For further discussion of the rules that govern when and how a destructor is executed, see §3.9.

The output of the example

using System;

class A  
{  
 ~A() {  
 Writeln("A's destructor");  
 }  
}

class B: A  
{  
 ~B() {  
 Writeln("B's destructor");  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 b = null;  
 GC.Collect();  
 GC.WaitForPendingFinalizers();  
 }  
}

is

B’s destructor  
A’s destructor

since destructors in an inheritance chain are called in order, from most derived to least derived.

Destructors are implemented by overriding the virtual method Finalize on System.Object. C# programs are not permitted to override this method or call it (or overrides of it) directly. For instance, the program

class A   
{  
 override protected void Finalize() {} // error

public void F() {  
 this.Finalize(); // error  
 }  
}

contains two errors.

The compiler behaves as if this method, and overrides of it, do not exist at all. Thus, this program:

class A   
{  
 void Finalize() {} // permitted  
}

is valid, and the method shown hides System.Object’s Finalize method.

For a discussion of the behavior when an exception is thrown from a destructor, see §16.3.

## Iterators

A function member (§7.5) implemented using an iterator block (§8.2) is called an iterator.

An iterator block may be used as the body of a function member as long as the return type of the corresponding function member is one of the enumerator interfaces (§10.14.1) or one of the enumerable interfaces (§10.14.2). It can occur as a method-body, operator-body or accessor-body, whereas events, instance constructors, static constructors and destructors cannot be implemented as iterators.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any ref or out parameters.

## 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 chapter these interfaces are referenced as IEnumerator and IEnumerator<T>, respectively.

## 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 chapter these interfaces are referenced as IEnumerable and IEnumerable<T>, respectively.

## 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.

## Enumerator objects

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an enumerator object is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object’s MoveNext method is invoked. An enumerator object has the following characteristics:

* It implements IEnumerator and IEnumerator<T>, where T is the yield type of the iterator.
* It implements System.IDisposable.
* It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
* It has four potential states, before, running, suspended, and after, and is initially in the before state.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (§2.4.2).

An enumerator object may implement more interfaces than those specified above.

The following sections describe the exact behavior of the MoveNext, Current, and Dispose members of the IEnumerable and IEnumerable<T> interface implementations provided by an enumerator object.

Note that enumerator objects do not support the IEnumerator.Reset method. Invoking this method causes a System.NotSupportedException to be thrown.

### The MoveNext method

The MoveNext method of an enumerator object encapsulates the code of an iterator block. Invoking the MoveNext method executes code in the iterator block and sets the Current property of the enumerator object as appropriate. The precise action performed by MoveNext depends on the state of the enumerator object when MoveNext is invoked:

* If the state of the enumerator object is before, invoking MoveNext:
* Changes the state to running.
* Initializes the parameters (including this) of the iterator block to the argument values and instance value saved when the enumerator object was initialized.
* Executes the iterator block from the beginning until execution is interrupted (as described below).
* If the state of the enumerator object is running, the result of invoking MoveNext is unspecified.
* If the state of the enumerator object is suspended, invoking MoveNext:
* Changes the state to running.
* Restores the values of all local variables and parameters (including this) to the values saved when execution of the iterator block was last suspended. Note that the contents of any objects referenced by these variables may have changed since the previous call to MoveNext.
* Resumes execution of the iterator block immediately following the yield return statement that caused the suspension of execution and continues until execution is interrupted (as described below).
* If the state of the enumerator object is after, invoking MoveNext returns false.

When MoveNext executes the iterator block, execution can be interrupted in four ways: By a yield return statement, by a yield break statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

* When a yield return statement is encountered (§8.14):
* The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the Current property of the enumerator object.
* Execution of the iterator body is suspended. The values of all local variables and parameters (including this) are saved, as is the location of this yield return statement. If the yield return statement is within one or more try blocks, the associated finally blocks are not executed at this time.
* The state of the enumerator object is changed to suspended.
* The MoveNext method returns true to its caller, indicating that the iteration successfully advanced to the next value.
* When a yield break statement is encountered (§8.14):
* If the yield break statement is within one or more try blocks, the associated finally blocks are executed.
* The state of the enumerator object is changed to after.
* The MoveNext method returns false to its caller, indicating that the iteration is complete.
* When the end of the iterator body is encountered:
* The state of the enumerator object is changed to after.
* The MoveNext method returns false to its caller, indicating that the iteration is complete.
* When an exception is thrown and propagated out of the iterator block:
* Appropriate finally blocks in the iterator body will have been executed by the exception propagation.
* The state of the enumerator object is changed to after.
* The exception propagation continues to the caller of the MoveNext method.

### 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.

### The Dispose method

The Dispose method is used to clean up the iteration by bringing the enumerator object to the after state.

* If the state of the enumerator object is before, invoking Dispose changes the state to after.
* If the state of the enumerator object is running, the result of invoking Dispose is unspecified.
* If the state of the enumerator object is suspended, invoking Dispose:
* Changes the state to running.
* Executes any finally blocks as if the last executed yield return statement were a yield break statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to after and the exception is propagated to the caller of the Dispose method.
* Changes the state to after.
* If the state of the enumerator object is after, invoking Dispose has no affect.

### Enumerable objects

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an enumerable object is created and returned. The enumerable object’s GetEnumerator method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object’s MoveNext method is invoked. An enumerable object has the following characteristics:

* It implements IEnumerable and IEnumerable<T>, where T is the yield type of the iterator.
* It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (§2.4.2).

An enumerable object may implement more interfaces than those specified above. In particular, an enumerable object may also implement IEnumerator and IEnumerator<T>, enabling it to serve as both an enumerable and an enumerator. In that type of implementation, the first time an enumerable object’s GetEnumerator method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object’s GetEnumerator, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another.

### The GetEnumerator method

An enumerable object provides an implementation of the GetEnumerator methods of the IEnumerable and IEnumerable<T> interfaces. The two GetEnumerator methods share a common implementation that acquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in §10.14.4.

## Implementation example

This section describes a possible implementation of iterators in terms of standard C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation or the only one possible.

The following Stack<T> class implements its GetEnumerator method using an iterator. The iterator enumerates the elements of the stack in top to bottom order.

using System;  
using System.Collections;  
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>  
{  
 T[] items;  
 int count;

public void Push(T item) {  
 if (items == null) {  
 items = new T[4];  
 }  
 else if (items.Length == count) {  
 T[] newItems = new T[count \* 2];  
 Array.Copy(items, 0, newItems, 0, count);  
 items = newItems;  
 }  
 items[count++] = item;  
 }

public T Pop() {  
 T result = items[--count];  
 items[count] = default(T);  
 return result;  
 }

public IEnumerator<T> GetEnumerator() {  
 for (int i = count - 1; i >= 0; --i) yield return items[i];  
 }  
}

The GetEnumerator method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

class Stack<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1: IEnumerator<T>, IEnumerator  
 {  
 int \_\_state;  
 T \_\_current;  
 Stack<T> \_\_this;  
 int i;

public \_\_Enumerator1(Stack<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 switch (\_\_state) {  
 case 1: goto \_\_state1;  
 case 2: goto \_\_state2;  
 }  
 i = \_\_this.count - 1;  
 \_\_loop:  
 if (i < 0) goto \_\_state2;  
 \_\_current = \_\_this.items[i];  
 \_\_state = 1;  
 return true;  
 \_\_state1:  
 --i;  
 goto \_\_loop;  
 \_\_state2:  
 \_\_state = 2;  
 return false;  
 }

public void Dispose() {  
 \_\_state = 2;  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

In the preceding translation, the code in the iterator block is turned into a state machine and placed in the MoveNext method of the enumerator class. Furthermore, the local variable i is turned into a field in the enumerator object so it can continue to exist across invocations of MoveNext.

The following example prints a simple multiplication table of the integers 1 through 10. The FromTo method in the example returns an enumerable object and is implemented using an iterator.

using System;  
using System.Collections.Generic;

class Test  
{  
 static IEnumerable<int> FromTo(int from, int to) {  
 while (from <= to) yield return from++;  
 }

static void Main() {  
 IEnumerable<int> e = FromTo(1, 10);  
 foreach (int x in e) {  
 foreach (int y in e) {  
 Console.Write("{0,3} ", x \* y);  
 }  
 Writeln();  
 }  
 }  
}

The FromTo method can be translated into an instantiation of a compiler-generated enumerable class that encapsulates the code in the iterator block, as shown in the following.

using System;  
using System.Threading;  
using System.Collections;  
using System.Collections.Generic;

class Test  
{  
 ...

static IEnumerable<int> FromTo(int from, int to) {  
 return new \_\_Enumerable1(from, to);  
 }

class \_\_Enumerable1:  
 IEnumerable<int>, IEnumerable,  
 IEnumerator<int>, IEnumerator  
 {  
 int \_\_state;  
 int \_\_current;  
 int \_\_from;  
 int from;  
 int to;  
 int i;

public \_\_Enumerable1(int \_\_from, int to) {  
 this.\_\_from = \_\_from;  
 this.to = to;  
 }

public IEnumerator<int> GetEnumerator() {  
 \_\_Enumerable1 result = this;  
 if (Interlocked.CompareExchange(ref \_\_state, 1, 0) != 0) {  
 result = new \_\_Enumerable1(\_\_from, to);  
 result.\_\_state = 1;  
 }  
 result.from = result.\_\_from;  
 return result;  
 }

IEnumerator IEnumerable.GetEnumerator() {  
 return (IEnumerator)GetEnumerator();  
 }

public int Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 switch (\_\_state) {  
 case 1:  
 if (from > to) goto case 2;  
 \_\_current = from++;  
 \_\_state = 1;  
 return true;  
 case 2:  
 \_\_state = 2;  
 return false;  
 default:  
 throw new InvalidOperationException();  
 }  
 }

public void Dispose() {  
 \_\_state = 2;  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

The enumerable class implements both the enumerable interfaces and the enumerator interfaces, enabling it to serve as both an enumerable and an enumerator. The first time the GetEnumerator method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object’s GetEnumerator, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The Interlocked.CompareExchange method is used to ensure thread-safe operation.

The from and to parameters are turned into fields in the enumerable class. Because from is modified in the iterator block, an additional \_\_from field is introduced to hold the initial value given to from in each enumerator.

The MoveNext method throws an InvalidOperationException if it is called when \_\_state is 0. This protects against use of the enumerable object as an enumerator object without first calling GetEnumerator.

The following example shows a simple tree class. The Tree<T> class implements its GetEnumerator method using an iterator. The iterator enumerates the elements of the tree in infix order.

using System;  
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>  
{  
 T value;  
 Tree<T> left;  
 Tree<T> right;

public Tree(T value, Tree<T> left, Tree<T> right) {  
 this.value = value;  
 this.left = left;  
 this.right = right;  
 }

public IEnumerator<T> GetEnumerator() {  
 if (left != null) foreach (T x in left) yield x;  
 yield value;  
 if (right != null) foreach (T x in right) yield x;  
 }  
}

class Program  
{  
 static Tree<T> MakeTree<T>(T[] items, int left, int right) {  
 if (left > right) return null;  
 int i = (left + right) / 2;  
 return new Tree<T>(items[i],   
 MakeTree(items, left, i - 1),  
 MakeTree(items, i + 1, right));  
 }

static Tree<T> MakeTree<T>(params T[] items) {  
 return MakeTree(items, 0, items.Length - 1);  
 }

// The output of the program is:  
 // 1 2 3 4 5 6 7 8 9  
 // Mon Tue Wed Thu Fri Sat Sun

static void Main() {  
 Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);  
 foreach (int i in ints) Console.Write("{0} ", i);  
 Writeln();

Tree<string> strings = MakeTree(  
 "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");  
 foreach (string s in strings) Console.Write("{0} ", s);  
 Writeln();  
 }  
}

The GetEnumerator method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

class Tree<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1 : IEnumerator<T>, IEnumerator  
 {  
 Node<T> \_\_this;  
 IEnumerator<T> \_\_left, \_\_right;  
 int \_\_state;  
 T \_\_current;

public \_\_Enumerator1(Node<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 try {  
 switch (\_\_state) {

case 0:  
 \_\_state = -1;  
 if (\_\_this.left == null) goto \_\_yield\_value;  
 \_\_left = \_\_this.left.GetEnumerator();  
 goto case 1;

case 1:  
 \_\_state = -2;  
 if (!\_\_left.MoveNext()) goto \_\_left\_dispose;  
 \_\_current = \_\_left.Current;  
 \_\_state = 1;  
 return true;

\_\_left\_dispose:  
 \_\_state = -1;  
 \_\_left.Dispose();

\_\_yield\_value:  
 \_\_current = \_\_this.value;  
 \_\_state = 2;  
 return true;

case 2:  
 \_\_state = -1;  
 if (\_\_this.right == null) goto \_\_end;  
 \_\_right = \_\_this.right.GetEnumerator();  
 goto case 3;

case 3:  
 \_\_state = -3;  
 if (!\_\_right.MoveNext()) goto \_\_right\_dispose;  
 \_\_current = \_\_right.Current;  
 \_\_state = 3;  
 return true;

\_\_right\_dispose:  
 \_\_state = -1;  
 \_\_right.Dispose();

\_\_end:  
 \_\_state = 4;  
 break;

}  
 }  
 finally {  
 if (\_\_state < 0) Dispose();  
 }  
 return false;  
 }

public void Dispose() {  
 try {  
 switch (\_\_state) {

case 1:  
 case -2:  
 \_\_left.Dispose();  
 break;

case 3:  
 case -3:  
 \_\_right.Dispose();  
 break;

}  
 }  
 finally {  
 \_\_state = 4;  
 }  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

The compiler generated temporaries used in the foreach statements are lifted into the \_\_left and \_\_right fields of the enumerator object. The \_\_state field of the enumerator object is carefully updated so that the correct Dispose() method will be called correctly if an exception is thrown. Note that it is not possible to write the translated code with simple foreach statements.

13

# INTERFACES

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

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

## Interface declarations

An interface-declaration is a type-declaration (§9.6) that declares a new interface type.

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface   
 identifier variant-type-parameter-listopt interface-baseopt  
 type-parameter-constraints-clausesopt interface-body ;opt

An interface-declaration consists of an optional set of attributes (§17), followed by an optional set of interface-modifiers (§13.1.1), followed by an optional partial modifier, followed by the keyword interface and an identifier that names the interface, followed by an optional variant-*type-parameter-list* specification (§13.1.3), followed by an optional interface-base specification (§13.1.4), followed by an optional type-parameter-constraints-clauses specification (§10.1.5), followed by an interface-body (§13.1.5), optionally followed by a semicolon.

### 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 §10.3.4.

The public, protected, internal, and private modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted (§3.5.1).

### Partial modifier

The partial modifier indicates that this interface-declaration is a partial type declaration. Multiple partial interface declarations with the same name within an enclosing namespace or type declaration combine to form one interface declaration, following the rules specified in §10.2.

### Variant type parameter lists

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.

variant-type-parameter-list:  
< variant-type-parameters >

variant-type-parameters:  
attributesopt variance-annotationopt  type-parameter  
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:  
in  
out

If the variance annotation is out, the type parameter is said to be covariant. If the variance annotation is in, the type parameter is said to be contravariant. If there is no variance annotation, the type parameter is said to be invariant.

In the example

interface C<out X, in Y, Z>   
{  
 X M(Y y);

Z P { get; set; }  
}

X is covariant, Y is contravariant and Z is invariant.

#### 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<A1,… AK> constructed from a generic type S<X1, .. XK> where for at least one Ai one of the following holds:
* Xi is covariant or invariant and Ai is output-unsafe.
* Xi is contravariant or invariant and Ai is input-safe.

A type T is input-unsafe if one of the following holds:

* T is a covariant type parameter
* T is an array type with an input-unsafe element type
* T is an interface or delegate type S<A1,… AK> constructed from a generic type S<X1, .. XK> where for at least one Ai one of the following holds:
* Xi is covariant or invariant and Ai is input-unsafe.
* Xi is contravariant or invariant and Ai 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.

#### Variance conversion

The purpose of variance annotations is to provide for more lenient (but still type safe) conversions to interface and delegate types. To this end the definitions of implicit (§6.1) and explicit conversions (§6.2) make use of the notion of variance-convertibility, which is defined as follows:

A type T<A1, …, An> is variance-convertible to a type T<B1, …, Bn> if T is either an interface or a delegate type declared with the variant type parameters T<X1, …, Xn>, and for each variant type parameter Xi one of the following holds:

* Xi is covariant and an implicit reference or identity conversion exists from Ai to Bi
* Xi is contravariant and an implicit reference or identity conversion exists from Bi to Ai
* Xi is invariant and an identity conversion exists from Ai to Bi

### Base interfaces

An interface can inherit from zero or more interface types, which are called the explicit base interfaces of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma separated list of base interface types.

interface-base:  
: interface-type-list

For a constructed interface type, the explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each type-parameter in the base interface declaration, the corresponding type-argument of the constructed type.

The explicit base interfaces of an interface must be at least as accessible as the interface itself (§3.5.4). For example, it is a compile-time error to specify a private or internal interface in the interface-base of a public interface.

It is a compile-time error for an interface to directly or indirectly inherit from itself.

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. In the example

interface IControl  
{  
 void Paint();  
}

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

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

interface IComboBox: ITextBox, IListBox {}

the base interfaces of IComboBox are IControl, ITextBox, and IListBox.

In other words, the IComboBox interface above inherits members SetText and SetItems as well as Paint.

Every base interface of an interface must be output-safe (§13.1.3.1). A class or struct that implements an interface also implicitly implements all of the interface’s base interfaces.

### Interface body

The interface-body of an interface defines the members of the interface.

interface-body:  
{ interface-member-declarationsopt }

## Interface members

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

interface-member-declarations:  
interface-member-declaration  
interface-member-declarations interface-member-declaration

interface-member-declaration:  
interface-method-declaration  
interface-property-declaration  
interface-event-declaration  
interface-indexer-declaration

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers abstract, public, protected, internal, private, virtual, override, or static.

The example

public delegate void StringListEvent(IStringList sender);

public interface IStringList  
{  
 void Add(string s);

int Count { get; }

event StringListEvent Changed;

string this[int index] { get; set; }  
}

declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An interface-declaration creates a new declaration space (§3.3), and the interface-member-declarations immediately contained by the interface-declaration introduce new members into this declaration space. The following rules apply to interface-member-declarations:

* The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature (§3.6) of a method must differ from the signatures of all other methods declared in the same interface, and two methods declared in the same interface may not have signatures that differ solely by ref and out.
* The name of a property or event must differ from the names of all other members declared in the same interface.
* The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a new modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in §3.7.1.2.

If a new modifier is included in a declaration that doesn’t hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

Note that the members in class object are not, strictly speaking, members of any interface (§13.2). However, the members in class object are available via member lookup in any interface type (§7.4).

### Interface methods

Interface methods are declared using interface-method-declarations:

interface-method-declaration:  
attributesopt newopt return-type identifier type-parameter-list  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

The attributes, return-type, identifier, and formal-parameter-list of an interface method declaration have the same meaning as those of a method declaration in a class (§10.6). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

Each formal parameter type of an interface method must be input-safe (§13.1.3.1), and the return type must be either void or output-safe. Furthermore, each class type constraint, interface type constraint and type parameter constraint on any type parameter of the method must be input-safe.

These rules ensure that any covariant or contravariant usage of the interface remains typesafe. For example,

interface I<out T> { void M<U>() where U : T; }

is illegal because the usage of T as a type parameter constraint on U is not input-safe.

Were this restriction not in place it would be possible to violate type safety in the following manner:

class B {}  
class D : B {}  
class E : B {}  
class C : I<D> { public void M<U>() {…} }  
…  
I<B> b = new C();  
b.M<E>();

This is actually a call to C.M<E>. But that call requires that E derive from D, so type safety would be violated here.

### Interface properties

Interface properties are declared using interface-property-declarations:

interface-property-declaration:  
attributesopt newopt type identifier { interface-accessors }

interface-accessors:  
attributesopt get ;  
attributesopt set ;  
attributesopt get ; attributesopt set ;  
attributesopt set ; attributesopt get ;

The attributes, type, and identifier of an interface property declaration have the same meaning as those of a property declaration in a class (§10.7).

The accessors of an interface property declaration correspond to the accessors of a class property declaration (§10.7.2), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

The type of an interface property must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

### Interface events

Interface events are declared using interface-event-declarations:

interface-event-declaration:  
attributesopt newopt event type identifier ;

The attributes, type, and identifier of an interface event declaration have the same meaning as those of an event declaration in a class (§10.8).

The type of an interface event must be input-safe.

### Interface indexers

Interface indexers are declared using interface-indexer-declarations:

interface-indexer-declaration:  
attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

The attributes, type, and formal-parameter-list of an interface indexer declaration have the same meaning as those of an indexer declaration in a class (§10.9).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (§10.9), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

All the formal parameter types of an interface indexer must be input-safe . In addition, any out or ref formal parameter types must also be output-safe. Note that even out parameters are required to be input-safe, due to a limitiation of the underlying execution platform.

The type of an interface indexer must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

### Interface member access

Interface members are accessed through member access (§7.6.4) and indexer access (§7.6.6.2) expressions of the form I.M and I[A], where I is an interface type, M is a method, property, or event of that interface type, and A is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup (§7.4), method invocation (§7.6.5.1), and indexer access (§7.6.6.2) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be used to resolve the ambiguities.

In the example

interface IList  
{  
 int Count { get; set; }  
}

interface ICounter  
{  
 void Count(int i);  
}

interface IListCounter: IList, ICounter {}

class C  
{  
 void Test(IListCounter x) {  
 x.Count(1); // Error  
 x.Count = 1; // Error  
 ((IList)x).Count = 1; // Ok, invokes IList.Count.set  
 ((ICounter)x).Count(1); // Ok, invokes ICounter.Count  
 }  
}

the first two statements cause compile-time errors because the member lookup (§7.4) of Count in IListCounter is ambiguous. As illustrated by the example, the ambiguity is resolved by casting x to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time.

In the example

interface IInteger  
{  
 void Add(int i);  
}

interface IDouble  
{  
 void Add(double d);  
}

interface INumber: IInteger, IDouble {}

class C  
{  
 void Test(INumber n) {  
 n.Add(1); // Invokes IInteger.Add  
 n.Add(1.0); // Only IDouble.Add is applicable  
 ((IInteger)n).Add(1); // Only IInteger.Add is a candidate  
 ((IDouble)n).Add(1); // Only IDouble.Add is a candidate  
 }  
}

the invocation n.Add(1) selects IInteger.Add by applying the overload resolution rules of §7.5.3. Similarly the invocation n.Add(1.0) selects IDouble.Add. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.

In the example

interface IBase  
{  
 void F(int i);  
}

interface ILeft: IBase  
{  
 new void F(int i);  
}

interface IRight: IBase  
{  
 void G();  
}

interface IDerived: ILeft, IRight {}

class A  
{  
 void Test(IDerived d) {  
 d.F(1); // Invokes ILeft.F  
 ((IBase)d).F(1); // Invokes IBase.F  
 ((ILeft)d).F(1); // Invokes ILeft.F  
 ((IRight)d).F(1); // Invokes IBase.F  
 }  
}

the IBase.F member is hidden by the ILeft.F member. The invocation d.F(1) thus selects ILeft.F, even though IBase.F appears to not be hidden in the access path that leads through IRight.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from IDerived to ILeft to IBase hides IBase.F, the member is also hidden in the access path from IDerived to IRight to IBase.

## Fully qualified interface member names

An interface member is sometimes referred to by its fully qualified name. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. For example, given the declarations

interface IControl  
{  
 void Paint();  
}

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

the fully qualified name of Paint is IControl.Paint and the fully qualified name of SetText is ITextBox.SetText.

In the example above, it is not possible to refer to Paint as ITextBox.Paint.

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. For example

namespace System  
{  
 public interface ICloneable  
 {  
 object Clone();  
 }  
}

Here, the fully qualified name of the Clone method is System.ICloneable.Clone.

## Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

interface ICloneable  
{  
 object Clone();  
}

interface IComparable  
{  
 int CompareTo(object other);  
}

class ListEntry: ICloneable, IComparable  
{  
 public object Clone() {...}

public int CompareTo(object other) {...}  
}

A class or struct that directly implements an interface also directly implements all of the interface’s base interfaces implicitly. This is true even if the class or struct doesn’t explicitly list all base interfaces in the base class list. For example:

interface IControl  
{  
 void Paint();  
}

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

class TextBox: ITextBox  
{  
 public void Paint() {...}

public void SetText(string text) {...}  
}

Here, class TextBox implements both IControl and ITextBox.

When a class C directly implements an interface, all classes derived from C also implement the interface implicitly. The base interfaces specified in a class declaration can be constructed interface types (§4.4). A base interface cannot be a type parameter on its own, though it can involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types:

class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in §13.4.2.

### Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare explicit interface member implementations. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example

interface IList<T>  
{  
 T[] GetElements();  
}

interface IDictionary<K,V>  
{  
 V this[K key];

void Add(K key, V value);  
}

class List<T>: IList<T>, IDictionary<int,T>  
{  
 T[] IList<T>.GetElements() {...}

T IDictionary<int,T>.this[int index] {...}

void IDictionary<int,T>.Add(int index, T value) {...}  
}

Here IDictionary<int,T>.this and IDictionary<int,T>.Add are explicit interface member implementations.

In some cases, the name of an interface member may not be appropriate for the implementing class, in which case the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a Close member function that has the effect of releasing the file resource, and implement the Dispose method of the IDisposable interface using explicit interface member implementation:

interface IDisposable  
{  
 void Dispose();  
}

class MyFile: IDisposable  
{  
 void IDisposable.Dispose() {  
 Close();  
 }

public void Close() {  
 // Do what's necessary to close the file  
 System.GC.SuppressFinalize(this);  
 }  
}

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can 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 access modifiers, and it is a compile-time error to include the modifiers abstract, virtual, override, or static.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through their fully qualified name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense also public.

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.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the following class

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}

int IComparable.CompareTo(object other) {...} // invalid  
}

the declaration of IComparable.CompareTo results in a compile-time error because IComparable is not listed in the base class list of Shape and is not a base interface of ICloneable. Likewise, in the declarations

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}  
}

class Ellipse: Shape  
{  
 object ICloneable.Clone() {...} // invalid  
}

the declaration of ICloneable.Clone in Ellipse results in a compile-time error because ICloneable is not explicitly listed in the base class list of Ellipse.

The fully qualified name of an interface member must reference the interface in which the member was declared. Thus, in the declarations

interface IControl  
{  
 void Paint();  
}

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

class TextBox: ITextBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}  
}

the explicit interface member implementation of Paint must be written as IControl.Paint.

### Uniqueness of implemented interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows:

interface I<T>  
{  
 void F();  
}

class X<U,V>: I<U>, I<V> // Error: I<U> and I<V> conflict  
{  
 void I<U>.F() {...}  
 void I<V>.F() {...}  
}

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();

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

* Let L be the list of interfaces directly specified in a generic class, struct, or interface declaration C.
* Add to L any base interfaces of the interfaces already in L.
* Remove any duplicates from L.
* If any possible constructed type created from C would, after type arguments are substituted into L, cause two interfaces in L to be identical, then the declaration of C is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration X above, the interface list L consists of I<U> and I<V>. The declaration is invalid because any constructed type with U and V being the same type would cause these two interfaces to be identical types.

It is possible for interfaces specified at different inheritance levels to unify:

interface I<T>  
{  
 void F();  
}

class Base<U>: I<U>  
{  
 void I<U>.F() {…}  
}

class Derived<U,V>: Base<U>, I<V> // 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> (§13.4.6).

### Implementation of generic methods

When a generic method implicitly implements an interface method, the constraints given for each method type parameter must be equivalent in both declarations (after any interface type parameters are replaced with the appropriate type arguments), where method type parameters are identified by ordinal positions, left to right.

When a generic method explicitly implements an interface method, however, no constraints are allowed on the implementing method. Instead, the constraints are inherited from the interface method

interface I<A,B,C>  
{  
 void F<T>(T t) where T: A;  
 void G<T>(T t) where T: B;  
 void H<T>(T t) where T: C;  
}

class C: I<object,C,string>  
{  
 public void F<T>(T t) {...} // 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 example, the explicit interface member implementation invokes a public method having strictly weaker constraints. Note that the assignment from t to s is valid since T inherits a constraint of T: string, even though this constraint is not expressible in source code.

### Interface mapping

A class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.

Interface mapping for a class or struct C locates an implementation for each member of each interface specified in the base class list of C. The implementation of a particular interface member I.M, where I is the interface in which the member M is declared, is determined by examining each class or struct S, starting with C and repeating for each successive base class of C, until a match is located:

* If S contains a declaration of an explicit interface member implementation that matches I and M, then this member is the implementation of I.M.
* Otherwise, if S contains a declaration of a 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. Note that the members of an interface include those members that are inherited from base interfaces.

For purposes of interface mapping, a class member A matches an interface member B when:

* A and B are methods, and the name, type, and formal parameter lists of A and B are identical.
* A and B are properties, the name and type of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).
* A and B are events, and the name and type of A and B are identical.
* A and B are indexers, the type and formal parameter lists of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are:

* Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
* Neither non-public nor static members participate in interface mapping.

In the example

interface ICloneable  
{  
 object Clone();  
}

class C: ICloneable  
{  
 object ICloneable.Clone() {...}

public object Clone() {...}  
}

the ICloneable.Clone member of C becomes the implementation of Clone in ICloneable because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. For example

interface IControl  
{  
 void Paint();  
}

interface IForm  
{  
 void Paint();  
}

class Page: IControl, IForm  
{  
 public void Paint() {...}  
}

Here, the Paint methods of both IControl and IForm are mapped onto the Paint method in Page. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For example

interface IBase  
{  
 int P { get; }  
}

interface IDerived: IBase  
{  
 new int P();  
}

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

class C: IDerived  
{  
 int IBase.P { get {...} }

int IDerived.P() {...}  
}

class C: IDerived  
{  
 public int P { get {...} }

int IDerived.P() {...}  
}

class C: IDerived  
{  
 int IBase.P { get {...} }

public int P() {...}  
}

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. In the example

interface IControl  
{  
 void Paint();  
}

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

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

class ComboBox: IControl, ITextBox, IListBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}

void IListBox.SetItems(string[] items) {...}  
}

it is not possible to have separate implementations for the IControl named in the base class list, the IControl inherited by ITextBox, and the IControl inherited by IListBox. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of ITextBox and IListBox share the same implementation of IControl, and ComboBox is simply considered to implement three interfaces, IControl, ITextBox, and IListBox.

The members of a base class participate in interface mapping. In the example

interface Interface1  
{  
 void F();  
}

class Class1  
{  
 public void F() {}

public void G() {}  
}

class Class2: Class1, Interface1  
{  
 new public void G() {}  
}

the method F in Class1 is used in Class2's implementation of Interface1.

### 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. For example, in the declarations

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public void Paint() {...}  
}

class TextBox: Control  
{  
 new public void Paint() {...}  
}

the Paint method in TextBox hides the Paint method in Control, but it does not alter the mapping of Control.Paint onto IControl.Paint, and calls to Paint through class instances and interface instances will have the following effects

Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint(); // invokes Control.Paint();  
t.Paint(); // invokes TextBox.Paint();  
ic.Paint(); // invokes Control.Paint();  
it.Paint(); // invokes Control.Paint();

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the declarations above to

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public virtual void Paint() {...}  
}

class TextBox: Control  
{  
 public override void Paint() {...}  
}

the following effects will now be observed

Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint(); // invokes Control.Paint();  
t.Paint(); // invokes TextBox.Paint();  
ic.Paint(); // invokes Control.Paint();  
it.Paint(); // invokes TextBox.Paint();

Since explicit interface member implementations cannot be declared 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. For 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.

### 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. For example, in the declarations

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() {...}  
}

class MyControl: Control, IControl  
{  
 public void Paint() {}  
}

the fact that Control maps IControl.Paint onto Control.IControl.Paint doesn’t affect the re-implementation in MyControl, which maps IControl.Paint onto MyControl.Paint.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. For example

interface IMethods  
{  
 void F();  
 void G();  
 void H();  
 void I();  
}

class Base: IMethods  
{  
 void IMethods.F() {}  
 void IMethods.G() {}  
 public void H() {}  
 public void I() {}  
}

class Derived: Base, IMethods  
{  
 public void F() {}  
 void IMethods.H() {}  
}

Here, the implementation of IMethods in Derived maps the interface methods onto Derived.F, Base.IMethods.G, Derived.IMethods.H, and Base.I.

When a class implements an interface, it implicitly also implements all of that interface’s base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface’s base interfaces. For example

interface IBase  
{  
 void F();  
}

interface IDerived: IBase  
{  
 void G();  
}

class C: IDerived  
{  
 void IBase.F() {...}

void IDerived.G() {...}  
}

class D: C, IDerived  
{  
 public void F() {...}

public void G() {...}  
}

Here, the re-implementation of IDerived also re-implements IBase, mapping IBase.F onto D.F.

### Abstract classes and interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. For example

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 public abstract void F();  
 public abstract void G();  
}

Here, the implementation of IMethods maps F and G onto abstract methods, which must be overridden in non-abstract classes that derive from C.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. For example

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 void IMethods.F() { FF(); }

void IMethods.G() { GG(); }

protected abstract void FF();

protected abstract void GG();  
}

Here, non-abstract classes that derive from C would be required to override FF and GG, thus providing the actual implementation of IMethods

14

# STRUCTS AND UNIONS

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.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

As described in §4.1.4, the simple types provided by C#, such as int, double, and bool, are in fact all struct types. Just as these predefined types are structs, it is also possible to use structs and operator overloading to implement new “primitive” types in the C# language. Two examples of such types are given at the end of this chapter (§11.4).

## Struct declarations

A struct-declaration is a type-declaration (§9.6) that declares a new struct:

struct-declaration:  
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
 struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

A struct-declaration consists of an optional set of attributes (§17), followed by an optional set of struct-modifiers (§11.1.1), followed by an optional partial modifier, followed by the keyword struct and an identifier that names the struct, followed by an optional type-parameter-list specification (§10.1.3), followed by an optional struct-interfaces specification (§11.1.2) ), followed by an optional type-parameters-constraints-clauses specification (§10.1.5), followed by a struct-body (§11.1.4), optionally followed by a semicolon.

### Struct modifiers

A struct-declaration may optionally include a sequence of struct modifiers:

struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier

struct-modifier:  
new  
public  
protected  
internal  
private

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration (§10.1).

### Partial modifier

The partial modifier indicates that this struct-declaration is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in §10.2.

### Struct interfaces

A struct declaration may include a struct-interfaces specification, in which case the struct is said to directly implement the given interface types.

struct-interfaces:  
: interface-type-list

Interface implementations are discussed further in §13.4.

### Struct body

The struct-body of a struct defines the members of the struct.

struct-body:  
{ struct-member-declarationsopt }

## 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

Except for the differences noted in §11.3, the descriptions of class members provided in §10.3 through §10.14 apply to struct members as well.

## Class and struct differences

Structs differ from classes in several important ways:

* Structs are value types (§11.3.1).
* All struct types implicitly inherit from the class System.ValueType (§11.3.2).
* Assignment to a variable of a struct type creates a copy of the value being assigned (§11.3.3).
* The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to null (§11.3.4).
* Boxing and unboxing operations are used to convert between a struct type and object (§11.3.5).
* The meaning of this is different for structs (§7.6.7).
* Instance field declarations for a struct are not permitted to include variable initializers (§11.3.7).
* A struct is not permitted to declare a parameterless instance constructor (§11.3.8).
* A struct is not permitted to declare a destructor (§11.3.9).

### Value semantics

Structs are value types (§4.1) and are said to have value semantics. Classes, on the other hand, are reference types (§4.2) 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 the data, the latter known as an object. When a struct B contains an instance field of type A and A is a struct type, it is a compile-time error for A to depend on B 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. For example

struct Node  
{  
 int data;

Node next; // error, Node directly depends on itself

}

is an error because Node contains an instance field of its own type. Another example

struct A { B b; }

struct B { C c; }

struct C { A a; }

is an error because each of the types A, B, and C depend on each other.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, 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, because structs are not reference types, it is not possible for values of a struct type to be null.

Given the declaration

struct Point  
{  
 public int x, y;

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

the code fragment

Point a = new Point(10, 10);  
Point b = a;  
a.x = 100;  
System.Console.WriteLine(b.x);

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.

### 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.

### 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 must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in §7.17.1.

### Default values

As described in §5.2, 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.

Referring to the Point struct declared above, the example

Point[] a = new Point[100];

initializes each Point in the array to the value produced by setting the x and y fields to zero.

The default value of a struct corresponds to the value returned by the default constructor of the struct (§4.1.2). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null.

Structs should be designed to consider the default initialization state a valid state. In the example

using System;

struct KeyValuePair  
{  
 string key;  
 string value;

public KeyValuePair(string key, string value) {  
 if (key == null || value == null) throw new ArgumentException();  
 this.key = key;  
 this.value = value;  
 }  
}

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a KeyValuePair variable is subject to default value initialization, the key and value fields will be null, and the struct must be prepared to handle this state.

### 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 type object or to an interface type that is implemented by the struct, a boxing operation takes place. Likewise, when a value of type object or a value of an interface type is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing copies the struct value either into or out of the boxed instance. Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct.

When a struct type overrides a virtual method inherited from System.Object (such as Equals, GetHashCode, or ToString), invocation of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example:

using System;

struct Counter  
{  
 int value;

public override string ToString() {  
 value++;  
 return value.ToString();  
 }  
}

class Program  
{  
 static void Test<T>() where T: new() {  
 T x = new T();  
 Console.WriteLine(x.ToString());  
 Console.WriteLine(x.ToString());  
 Console.WriteLine(x.ToString());  
 }

static void Main() {  
 Test<Counter>();  
 }  
}

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().

Similarly, boxing never implicitly occurs when accessing a member on a constrained type parameter. 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.

using System;

interface ICounter  
{  
 void Increment();  
}

struct Counter: ICounter  
{  
 int value;

public override string ToString() {  
 return value.ToString();  
 }

void ICounter.Increment() {  
 value++;  
 }  
}

class Program  
{  
 static void Test<T>() where T: ICounter, new() {  
 T x = new T();  
 Console.WriteLine(x);  
 x.Increment(); // Modify x  
 Console.WriteLine(x);  
 ((ICounter)x).Increment(); // Modify boxed copy of x  
 Console.WriteLine(x);  
 }

static void Main() {  
 Test<Counter>();  
 }  
}

The first call to Increment modifies the value in the variable x. This is not equivalent to the second call to Increment, which modifies the value in a boxed copy of x. Thus, the output of the program is:

0  
1  
1

For further details on boxing and unboxing, see §4.3.

### Meaning of this

Within an instance constructor or instance function member of a class, this is classified as a value. Thus, while this can be used to refer to the instance for which the function member was invoked, it is not possible to assign to this in a function member of a class.

Within an instance constructor of a struct, this corresponds to an out parameter of the struct type, and within an instance function member of a struct, this corresponds to a ref parameter of the struct type. In both cases, this is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to this or by passing this as a ref or out parameter.

### Field initializers

As described in §11.3.4, the default value of a struct consists of the value that results from setting all value type fields to their default 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.

The example

struct Point  
{  
 public int x = 1; // Error, initializer not permitted  
 public int y = 1; // Error, initializer not permitted  
}

is in error because the instance field declarations include variable initializers.

### 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 (§4.1.2). A struct can declare instance constructors having parameters. For example

struct Point  
{  
 int x, y;

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

Given the above declaration, the statements

Point p1 = new Point();

Point p2 = new Point(0, 0);

both create a Point with x and y initialized to zero.

A struct instance constructor is not permitted to include a constructor initializer of the form base(...).

If the struct instance constructor doesn’t specify a constructor initializer, the this variable corresponds to an out parameter of the struct type, and similar to an out parameter, this must be definitely assigned (§5.3) at every location where the constructor returns. If the struct instance constructor specifies a constructor initializer, the this variable corresponds to a ref parameter of the struct type, and similar to a ref parameter, this is considered definitely assigned on entry to the constructor body. Consider the instance constructor implementation below:

struct Point  
{  
 int x, y;

public int X {  
 set { x = value; }  
 }

public int Y {  
 set { y = value; }  
 }

public Point(int x, int y) {  
 X = x; // error, this is not yet definitely assigned  
 Y = y; // error, this is not yet definitely assigned  
 }  
}

No instance member function (including the set accessors for the properties X and Y) can be called until all fields of the struct being constructed have been definitely assigned. Note, however, that if Point were a class instead of a struct, the instance constructor implementation would be permitted.

### Destructors

A struct is not permitted to declare a destructor.

### Static constructors

Static constructors for structs follow most of the same rules as for classes. The execution of a static constructor for a struct type is triggered by the first of the following events to occur within an application domain:

* A static member of the struct type is referenced.
* An explicitly declared constructor of the struct type is called.

The creation of default values (§11.3.4) of struct types does not trigger the static constructor. (An example of this is the initial value of elements in an array.)

## Struct examples

The following shows two significant examples of using struct types to create types that can be used similarly to the predefined types of the language, but with modified semantics.

### Database integer type

The DBInt struct below implements an integer type that can represent the complete set of values of the int type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

using System;

public struct DBInt  
{  
 // The Null member represents an unknown DBInt value.

public static readonly DBInt Null = new DBInt();

// When the defined field is true, this DBInt represents a known value  
 // which is stored in the value field. When the defined field is false,  
 // this DBInt represents an unknown value, and the value field is 0.

int value;  
 bool defined;

// Private instance constructor. Creates a DBInt with a known value.

DBInt(int value) {  
 this.value = value;  
 this.defined = true;  
 }

// The IsNull property is true if this DBInt represents an unknown value.

public bool IsNull { get { return !defined; } }

// The Value property is the known value of this DBInt, or 0 if this  
 // DBInt represents an unknown value.

public int Value { get { return value; } }

// Implicit conversion from int to DBInt.

public static implicit operator DBInt(int x) {  
 return new DBInt(x);  
 }

// Explicit conversion from DBInt to int. Throws an exception if the  
 // given DBInt represents an unknown value.

public static explicit operator int(DBInt x) {  
 if (!x.defined) throw new InvalidOperationException();  
 return x.value;  
 }

public static DBInt operator +(DBInt x) {  
 return x;  
 }

public static DBInt operator -(DBInt x) {  
 return x.defined ? -x.value : Null;  
 }

public static DBInt operator +(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value + y.value: Null;  
 }

public static DBInt operator -(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value - y.value: Null;  
 }

public static DBInt operator \*(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value \* y.value: Null;  
 }

public static DBInt operator /(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value / y.value: Null;  
 }

public static DBInt operator %(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value % y.value: Null;  
 }

public static DBBool operator ==(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value == y.value: DBBool.Null;  
 }

public static DBBool operator !=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value != y.value: DBBool.Null;  
 }

public static DBBool operator >(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value > y.value: DBBool.Null;  
 }

public static DBBool operator <(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value < y.value: DBBool.Null;  
 }

public static DBBool operator >=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value >= y.value: DBBool.Null;  
 }

public static DBBool operator <=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value <= y.value: DBBool.Null;  
 }

public override bool Equals(object obj) {  
 if (!(obj is DBInt)) return false;  
 DBInt x = (DBInt)obj;  
 return value == x.value && defined == x.defined;  
 }

public override int GetHashCode() {  
 return value;  
 }

public override string ToString() {  
 return defined? value.ToString(): “DBInt.Null”;  
 }  
}

### Database boolean type

The DBBool struct below implements a three-valued logical type. The possible values of this type are DBBool.True, DBBool.False, and DBBool.Null, where the Null member indicates an unknown value. Such three-valued logical types are commonly used in databases.

using System;

public struct DBBool  
{  
 // The three possible DBBool values.

public static readonly DBBool Null = new DBBool(0);  
 public static readonly DBBool False = new DBBool(-1);  
 public static readonly DBBool True = new DBBool(1);

// Private field that stores –1, 0, 1 for False, Null, True.

sbyte value;

// Private instance constructor. The value parameter must be –1, 0, or 1.

DBBool(int value) {  
 this.value = (sbyte)value;  
 }

// Properties to examine the value of a DBBool. Return true if this  
 // DBBool has the given value, false otherwise.

public bool IsNull { get { return value == 0; } }

public bool IsFalse { get { return value < 0; } }

public bool IsTrue { get { return value > 0; } }

// Implicit conversion from bool to DBBool. Maps true to DBBool.True and  
 // false to DBBool.False.

public static implicit operator DBBool(bool x) {  
 return x? True: False;  
 }

// Explicit conversion from DBBool to bool. Throws an exception if the  
 // given DBBool is Null, otherwise returns true or false.

public static explicit operator bool(DBBool x) {  
 if (x.value == 0) throw new InvalidOperationException();  
 return x.value > 0;  
 }

// Equality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.

public static DBBool operator ==(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value == y.value? True: False;  
 }

// Inequality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.

public static DBBool operator !=(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value != y.value? True: False;  
 }

// Logical negation operator. Returns True if the operand is False, Null  
 // if the operand is Null, or False if the operand is True.

public static DBBool operator !(DBBool x) {  
 return new DBBool(-x.value);  
 }

// Logical AND operator. Returns False if either operand is False,  
 // otherwise Null if either operand is Null, otherwise True.

public static DBBool operator &(DBBool x, DBBool y) {  
 return new DBBool(x.value < y.value? x.value: y.value);  
 }

// Logical OR operator. Returns True if either operand is True, otherwise  
 // Null if either operand is Null, otherwise False.

public static DBBool operator |(DBBool x, DBBool y) {  
 return new DBBool(x.value > y.value? x.value: y.value);  
 }

// Definitely true operator. Returns true if the operand is True, false  
 // otherwise.

public static bool operator true(DBBool x) {  
 return x.value > 0;  
 }

// Definitely false operator. Returns true if the operand is False, false  
 // otherwise.

public static bool operator false(DBBool x) {  
 return x.value < 0;  
 }

public override bool Equals(object obj) {  
 if (!(obj is DBBool)) return false;  
 return value == ((DBBool)obj).value;  
 }

public override int GetHashCode() {  
 return value;  
 }

public override string ToString() {  
 if (value > 0) return "DBBool.True";  
 if (value < 0) return "DBBool.False";  
 return "DBBool.Null";  
 }  
}

15

# ENUMS

An enum type is a distinct value type (§4.1) that declares a set of named constants.

The example

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

declares an enum type named Color with members Red, Green, and Blue.

## 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:  
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:  
: integral-type

enum-body:  
{ enum-member-declarationsopt }  
{ enum-member-declarations , }

Each enum type has a corresponding integral type called the underlying type of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of byte, sbyte, short, ushort, int, uint, long or ulong. Note that char cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of int.

The example

enum Color: long  
{  
 Red,  
 Green,  
 Blue  
}

declares an enum with an underlying type of long. A developer might choose to use an underlying type of long, as in the example, to enable the use of values that are in the range of long but not in the range of int, or to preserve this option for the future.

## 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 (§10.1.1). Note, however, that the abstract and sealed modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

## 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:  
attributesopt identifier  
attributesopt 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 must be in the range of the underlying type for the enum. The example

enum Color: uint  
{  
 Red = -1,  
 Green = -2,  
 Blue = -3  
}

results in a compile-time error because the constant values -1, -2, and –3 are not in the range of the underlying integral type uint.

Multiple enum members may share the same associated value. The example

enum Color   
{  
 Red,  
 Green,  
 Blue,

Max = Blue  
}

shows an enum in which two enum members—Blue and Max—have the same associated value.

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a constant-expression initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

* If the enum member is the first enum member declared in the enum type, its associated value is 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 must be within the range of values that can be represented by the underlying type, otherwise a compile-time error occurs.

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.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

The example

enum Circular  
{  
 A = B,  
 B  
}

results in a compile-time error because the declarations of A and B are circular. A depends on B explicitly, and B depends on A implicitly.

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility—an enum member is accessible if its containing enum type is accessible.

## The System.Enum type

The type System.Enum is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from System.Enum are available in any enum type. A boxing conversion (§4.3.1) exists from any enum type to System.Enum, and an unboxing conversion (§4.3.2) exists from System.Enum to any enum type.

Note that System.Enum is not itself an enum-type. Rather, it is a class-type from which all enum-types are derived. The type System.Enum inherits from the type System.ValueType (§4.1.1), 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.

## Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion (§6.2.2) is required to convert between an enum type and an integral type, or between two enum types. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type.

Enum members have the type of their containing enum type (except within other enum member initializers: see §14.3). The value of an enum member declared in enum type E with associated value v is (E)v.

The following operators can be used on values of enum types: ==, !=, <, >, <=, >= (§7.10.5), binary + (§7.8.4), binary ‑ (§7.8.5), ^, &, | (§7.11.2), ~ (§7.7.4), ++ and -- (§7.6.9 and §7.7.5).

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.

16

# DELEGATES

Delegates enable scenarios that other languages—such as C++, Pascal, and Modula—have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that is derived from the class System.Delegate. A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate’s callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (§15.1) with the delegate’s type. This makes delegates perfectly suited for “anonymous” invocation.

## Delegate declarations

A delegate-declaration is a type-declaration (§9.6) that declares a new delegate type.

delegate-declaration:  
attributesopt delegate-modifiersopt delegate return-type   
 identifier variant-type-parameter-listopt   
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers:  
delegate-modifier  
delegate-modifiers delegate-modifier

delegate-modifier:  
new  
public  
protected  
internal  
private

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

The new modifier is 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 §10.3.4.

The public, protected, internal, and private modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted (§3.5.1).

The delegate’s type name is identifier.

The optional formal-parameter-list specifies the parameters of the delegate, and return-type indicates the return type of the delegate.

The optional variant-type-parameter-list (§13.1.3) specifies the type parameters to the delegate itself.

The return type of a delegate type must be either void, or output-safe (§13.1.3.1).

All the formal parameter types of a delegate type must be input-safe. Additionally, any out or ref parameter types must also be output-safe. Note that even out parameters are required to be input-safe, due to a limitiation of the underlying execution platform.

Delegate types in C# are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. However, instances of two distinct but structurally equivalent delegate types may compare as equal (§7.9.8).

For example:

delegate int D1(int i, double d);

class A  
{  
 public static int M1(int a, double b) {...}  
}

class B  
{  
 delegate int D2(int c, double d);

public static int M1(int f, double g) {...}

public static void M2(int k, double l) {...}

public static int M3(int g) {...}

public static void M4(int g) {...}  
}

The 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; however, these delegate types are two different types, so they are not interchangeable. 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.

Like other generic type declarations, type arguments must be given to create a constructed delegate type. The parameter types and return type of a constructed delegate type are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used in determining what methods are compatible with a constructed delegate type. For example:

delegate bool Predicate<T>(T value);

class X  
{  
 static bool F(int i) {...}

static bool G(string s) {...}  
}

The method X.F is compatible with the delegate type Predicate<int> and the method X.G is compatible with the delegate type Predicate<string> .

The only way to declare a delegate type is via a delegate-declaration. A delegate type is a class type that is derived from System.Delegate. Delegate types are implicitly sealed, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from System.Delegate. Note that System.Delegate is not itself a delegate type; it is a class type from which all delegate types are derived.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the System.Delegate type via the usual member access syntax.

The set of methods encapsulated by a delegate instance is called an invocation list. When a delegate instance is created (§15.2) from a single method, it encapsulates that method, and its invocation list contains only one entry. However, when two non-null delegate instances are combined, their invocation lists are concatenated—in the order left operand then right operand—to form a new invocation list, which contains two or more entries.

Delegates are combined using the binary + (§7.8.4) and += operators (§7.17.2). A delegate can be removed from a combination of delegates, using the binary - (§7.8.5) and -= operators (§7.17.2). Delegates can be compared for equality (§7.10.8).

The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

delegate void D(int x);

class C  
{  
 public static void M1(int i) {...}

public static void M2(int i) {...}

}

class Test  
{  
 static void Main() {  
 D cd1 = new D(C.M1); // M1  
 D cd2 = new D(C.M2); // M2  
 D cd3 = cd1 + cd2; // M1 + M2  
 D cd4 = cd3 + cd1; // M1 + M2 + M1  
 D cd5 = cd4 + cd3; // M1 + M2 + M1 + M1 + M2  
 }

}

When cd1 and cd2 are instantiated, they each encapsulate one method. When cd3 is instantiated, it has an invocation list of two methods, M1 and M2, in that order. cd4’s invocation list contains M1, M2, and M1, in that order. Finally, cd5’s invocation list contains M1, M2, M1, M1, and M2, in that order. For more examples of combining (as well as removing) delegates, see §15.4.

## Delegate compatibility

A method or delegate M is compatible with a delegate type D if all of the following are true:

* D and M have the same number of parameters, and each parameter in D has the same ref or out modifiers as the corresponding parameter in M.
* For each value parameter (a parameter with no ref or out modifier), an identity conversion (§6.1.1) or implicit reference conversion (§6.1.6) exists from the parameter type in D to the corresponding parameter type in M.
* For each ref or out parameter, the parameter type in D is the same as the parameter type in M.
* An identity or implicit reference conversion exists from the return type of M to the return type of D.

## Delegate instantiation

An instance of a delegate is created by a delegate-creation-expression (§7.6.10.5) or a conversion to a delegate type. The newly created delegate instance then refers to either:

* The static method referenced in the delegate-creation-expression, or
* The target object (which cannot be null) and instance method referenced in the delegate-creation-expression, or
* Another delegate.

For example:

delegate void D(int x);

class C  
{  
 public static void M1(int i) {...}  
 public void M2(int i) {...}  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1); // static method  
 C t = new C();  
 D cd2 = new D(t.M2); // instance method  
 D cd3 = new D(cd2); // another delegate  
 }  
}

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

## Delegate invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. (See §7.6.5.3 for detailed information on delegate invocation.) If an exception occurs during the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method to which that delegate referred.

Invocation of a delegate instance whose invocation list contains multiple entries proceeds by invoking each of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes reference parameters (§10.6.1.2), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

Attempting to invoke a delegate instance whose value is null results in an exception of type System.NullReferenceException.

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);  
 cd1(-1); // call M1

D cd2 = new D(C.M2);  
 cd2(-2); // call M2

D cd3 = cd1 + cd2;  
 cd3(10); // call M1 then M2

cd3 += cd1;  
 cd3(20); // call M1, M2, then M1

C c = new C();  
 D cd4 = new D(c.M3);  
 cd3 += cd4;  
 cd3(30); // call M1, M2, M1, then M3

cd3 -= cd1; // remove last M1  
 cd3(40); // call M1, M2, then M3

cd3 -= cd4;  
 cd3(50); // call M1 then M2

cd3 -= cd2;  
 cd3(60); // call M1

cd3 -= cd2; // impossible removal is benign  
 cd3(60); // call M1

cd3 -= cd1; // invocation list is empty so cd3 is null

// cd3(70); // System.NullReferenceException thrown

cd3 -= cd1; // impossible removal is benign  
 }  
}

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

18

# ARRAYS

An array is a data structure that contains a number of 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.

An array has a rank which 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 which 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.

The element type of an array can be any type, including an array type.

## Array types

An array type is written as a non-array-type followed by one or more rank-specifiers:

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

A non-array-type is any type that is not itself an array-type.

The rank of an array type is given by the leftmost rank-specifier in the array-type: A rank-specifier indicates that the array is an array with a rank of one plus the number of “,” tokens in the rank-specifier.

The element type of an array type is the type that results from deleting the leftmost rank-specifier:

* An array type of the form T[R] is an array with rank R and a non-array element type T.
* An array type of the form T[R][R1]...[RN] is an array with rank R and an element type T[R1]...[RN].

In effect, the rank-specifiers are read from left to right before the final non-array element type. The type int[][,,][,] is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of int.

At run-time, a value of an array type can be null or a reference to an instance of that array type.

### The System.Array type

The type System.Array is the abstract base type of all array types. An implicit reference conversion (§6.1.6) exists from any array type to System.Array, and an explicit reference conversion (§6.2.4) exists from System.Array to any array type. Note that System.Array is not itself an array-type. Rather, it is a class-type from which all array-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.

### Arrays and the generic IList interface

A one-dimensional array T[] implements the interface System.Collections.Generic.IList<T> (IList<T> for short) and its base interfaces. Accordingly, there is an implicit conversion from T[] to IList<T> and its base interfaces. In addition, if there is an implicit reference conversion from S to T then S[] implements IList<T> and there is an implicit reference conversion from S[] to IList<T> and its base interfaces (§6.1.6). If there is an explicit reference conversion from S to T then there is an explicit reference conversion from S[] to IList<T> and its base interfaces (§6.2.4). For example:

using System.Collections.Generic;

class Test  
{  
 static void Main() {  
 string[] sa = new string[5];  
 object[] oa1 = new object[5];  
 object[] oa2 = sa;

IList<string> lst1 = sa; // Ok  
 IList<string> lst2 = oa1; // Error, cast needed  
 IList<object> lst3 = sa; // Ok  
 IList<object> lst4 = oa1; // Ok

IList<string> lst5 = (IList<string>)oa1; // Exception  
 IList<string> lst6 = (IList<string>)oa2; // Ok  
 }  
}

The assignment lst2 = oa1 generates a compile-time error since the conversion from object[] to IList<string> is an explicit conversion, not implicit. The cast (IList<string>)oa1 will cause an exception to be thrown at run-time since oa1 references an object[] and not a string[]. However the cast (IList<string>)oa2 will not cause an exception to be thrown since oa2 references a string[].

Whenever there is an implicit or explicit reference conversion from S[] to IList<T>, there is also an explicit reference conversion from IList<T> and its base interfaces to S[] (§6.2.4).

When an array type S[] implements IList<T>, some of the members of the implemented interface may throw exceptions. The precise behavior of the implementation of the interface is beyond the scope of this specification.

## Array creation

Array instances are created by array-creation-expressions (§7.6.10.4) or by field or local variable declarations that include an array-initializer (§12.6).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The System.Array type is an abstract type that cannot be instantiated.

Elements of arrays created by array-creation-expressions are always initialized to their default value (§5.2).

## Array element access

Array elements are accessed using element-access expressions (§7.6.6.1) of the form A[I1, I2, ..., IN], where A is an expression of an array type and each IX is an expression of type int, uint, long, ulong, or can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a foreach statement (§8.8.4).

## Array members

Every array type inherits the members declared by the System.Array type.

## Array covariance

For any two reference-types A and B, if an implicit reference conversion (§6.1.6) or explicit reference conversion (§6.2.4) exists from A to B, then the same reference conversion also exists from the array type A[R] to the array type B[R], where R is any given rank-specifier (but the same for both array types). This relationship is known as array covariance. Array covariance in particular means that a value of an array type A[R] may actually be a reference to an instance of an array type B[R], provided an implicit reference conversion exists from B to A.

Because of array covariance, assignments to elements of reference type arrays include a run-time check which ensures that the value being assigned to the array element is actually of a permitted type (§7.17.1). For example:

class Test  
{  
 static void Fill(object[] array, int index, int count, object value) {  
 for (int i = index; i < index + count; i++) array[i] = value;  
 }

static void Main() {  
 string[] strings = new string[100];  
 Fill(strings, 0, 100, "Undefined");  
 Fill(strings, 0, 10, null);  
 Fill(strings, 90, 10, 0);  
 }  
}

The assignment to array[i] in the Fill method implicitly includes a run-time check which ensures that the object referenced by value is either null or an instance that is compatible with the actual element type of array. In Main, the first two invocations of Fill succeed, but the third invocation causes a System.ArrayTypeMismatchException to be thrown upon executing the first assignment to array[i]. The exception occurs because a boxed int cannot be stored in a string array.

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[].

## Array initializers

Array initializers may be specified in field declarations (§10.5), local variable declarations (§8.5.1), and array creation expressions (§7.6.10.4):

array-initializer:  
{ variable-initializer-listopt }  
{ variable-initializer-list , }

variable-initializer-list:  
variable-initializer  
variable-initializer-list , variable-initializer

variable-initializer:  
expression  
array-initializer

An array initializer consists of a sequence of variable initializers, enclosed by “{”and “}” tokens and separated by “,” tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer, or is inferred from the expressions in the array initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

int[] a = {0, 2, 4, 6, 8};

it is simply shorthand for an equivalent array creation expression:

int[] a = new int[] {0, 2, 4, 6, 8};

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an int[] instance of length 5 and then initializes the instance with the following values:

a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};

creates a two-dimensional array with a length of 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;

If a dimension other than the rightmost is given with length zero, the subsequent dimensions are assumed to also have length zero. The 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];

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Here are some examples:

int i = 3;  
int[] x = new int[3] {0, 1, 2}; // 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.

19

# exceptions

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences:

* In C#, all exceptions must be represented by an instance of a class type derived from System.Exception. In C++, any value of any type can be used to represent an exception.
* In C#, a finally block (§8.10) can be used to write termination code that executes in both normal execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.
* In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well defined exception classes and are on a par with application-level error conditions.

## Causes of exceptions

Exception can be thrown in two different ways.

* A throw statement (§8.9.5) throws an exception immediately and unconditionally. Control never reaches the statement immediately following the throw.
* Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation (§7.8.2) throws a System.DivideByZeroException if the denominator is zero. See §16.4 for a list of the various exceptions that can occur in this way.

## 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.

## How exceptions are handled

Exceptions are handled by a try statement (§8.10).

When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the 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 that encloses the point of the call to the current method. This search continues until a catch clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the 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, one of two things occurs:

* If the search for a matching catch clause reaches a static constructor (§10.12) or static field initializer, then a System.TypeInitializationException is thrown at the point that triggered the invocation of the static constructor. The inner exception of the System.TypeInitializationException contains the exception that was originally thrown.
* If the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution, and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the object type) or if there is no base class destructor, then the exception is discarded.

## 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 actual type of the stored element is incompatible with the actual type of the array. |
| System.DivideByZeroException | Thrown when an attempt to divide an integral value by zero occurs. |
| System.IndexOutOfRangeException | Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array. |
| 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 method calls; typically indicative of very deep or unbounded recursion. |
| System.TypeInitializationException | Thrown when a static constructor throws an exception, and no catch clauses exists to catch it. |

20

# STATEMENTS

C# provides a variety of statements. Most of these statements will be familiar to developers who have programmed in C and C++.

statement:  
labeled-statement  
declaration-statement  
embedded-statement

embedded-statement:  
block  
empty-statement  
expression-statement  
selection-statement  
iteration-statement  
jump-statement  
try-statement  
checked-statement  
unchecked-statement  
lock-statement  
using-statement   
yield-statement

The embedded-statement nonterminal is used for statements that appear within other statements. The use of embedded-statement rather than statement excludes the use of declaration statements and labeled statements in these contexts. The example

void F(bool b) {  
 if (b)  
 int i = 44;  
}

results in a compile-time error because an if statement requires an embedded-statement rather than a statement for its if branch. If this code were permitted, then the variable i 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 points and reachability

Every statement has an end point. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be reachable. Conversely, if there is no possibility that a statement will be executed, the statement is said to be unreachable.

In the example

void F() {  
 Console.WriteLine("reachable");  
 goto Label;  
 Console.WriteLine("unreachable");  
 Label:  
 Console.WriteLine("reachable");  
}

the second invocation of Console.WriteLine is unreachable because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (§7.19) that control the behavior of statements, but the possible values of 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.

The block of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

In the example

void F(int x) {  
 Console.WriteLine("start");  
 if (x < 0) Console.WriteLine("negative");  
}

the reachability of the second Console.WriteLine is determined as follows:

* The first Console.WriteLine expression statement is reachable because the block of the F method is reachable.
* The end point of the first Console.WriteLine expression statement is reachable because that statement is reachable.
* The if statement is reachable because the end point of the first Console.WriteLine expression statement is reachable.
* The second Console.WriteLine expression statement is reachable because the boolean expression of the if statement does not have the constant value false.

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

* Because the switch statement does not permit a switch section to “fall through” to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a break statement is missing.
* It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it typically is an indication that a return statement is missing.

## Blocks

A block permits multiple statements to be written in contexts where a single statement is allowed.

block:  
{ statement-listopt }

A block consists of an optional statement-list (§8.2.1), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements (§8.5). The scope of a local variable or constant declared in a block is the block.

Within a block, the meaning of a name used in an expression context must always be the same (§7.6.2.1).

A block is executed as follows:

* If the block is empty, control is transferred to the end point of the block.
* If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A block that contains one or more yield statements (§8.14) is called an iterator block. Iterator blocks are used to implement function members as iterators (§10.14). Some additional restrictions apply to iterator blocks:

* It is a compile-time error for a return statement to appear in an iterator block (but yield return statements are permitted).
* It is a compile-time error for an iterator block to contain an unsafe context (§18.1). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

### Statement lists

A statement list consists of one or more statements written in sequence. Statement lists occur in blocks (§8.2) and in switch-blocks (§8.7.2).

statement-list:  
statement  
statement-list statement

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

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.

## 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.

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: ;  
}

## Labeled statements

A labeled-statement permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

labeled-statement:  
identifier : statement

A labeled statement declares a label with the name given by the identifier. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from goto statements (§8.9.3) within the scope of the label. This means that goto statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

int F(int x) {  
 if (x >= 0) goto x;  
 x = -x;  
 x: return x;  
}

is valid and uses the name x as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable goto statement. (Exception: If a goto statement is inside a try that includes a finally block, and the labeled statement is outside the try, and the end point of the finally block is unreachable, then the labeled statement is not reachable from that goto statement.)

## Declaration statements

A declaration-statement declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

declaration-statement:  
local-variable-declaration ;  
local-constant-declaration ;

### 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 (§2.4.3).When the local-variable-type is specified as var and no type named var is in scope, the declaration is an implicitly typed local variable declaration, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

* The local-variable-declaration cannot include multiple local-variable-declarators.
* The local-variable-declarator must include a local-variable-initializer.
* The local-variable-initializer must be an expression.
* The initializer expression must have a compile-time type.
* The initializer expression cannot refer to the declared variable itself

The following are examples of incorrect implicitly typed local variable declarations:

var x; // Error, no initializer to infer type from  
var y = {1, 2, 3}; // Error, array initializer not permitted  
var z = null; // Error, null does not have a type  
var u = x => x + 1; // Error, anonymous functions do not have a type  
var v = v++; // Error, initializer cannot refer to variable itself

The value of a local variable is obtained in an expression using a simple-name (§7.6.2), and the value of a local variable is modified using an assignment (§7.17). A local variable must be definitely assigned (§5.3) at each location where its value is obtained.

The scope of a local variable declared in a local-variable-declaration is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the local-variable-declarator of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

void F() {  
 int x = 1, y, z = x \* 2;  
}

corresponds exactly to

void F() {  
 int x; x = 1;  
 int y;  
 int z; z = x \* 2;  
}

In an implicitly typed local variable declaration, the type of the local variable being declared is taken to be the same as the type of the expression used to initialize the variable. For example:

var i = 5;  
var s = "Hello";  
var d = 1.0;  
var numbers = new int[] {1, 2, 3};  
var orders = new Dictionary<int,Order>();

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

int i = 5;  
string s = "Hello";  
double d = 1.0;  
int[] numbers = new int[] {1, 2, 3};  
Dictionary<int,Order> orders = new Dictionary<int,Order>();

### 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 (§7.19) that gives the value of the constant.

The type and constant-expression of a local constant declaration must follow the same rules as those of a constant member declaration (§10.4).

The value of a local constant is obtained in an expression using a simple-name (§7.6.2).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its constant-declarator. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

## Expression statements

An expression-statement evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement:  
statement-expression ;

statement-expression:  
invocation-expression  
object-creation-expression  
assignment  
post-increment-expression  
post-decrement-expression  
pre-increment-expression  
pre-decrement-expression  
await-expression

Not all expressions are permitted as statements. In particular, expressions such as x + y and x == 1 that merely compute a value (which will be discarded), are not permitted as statements.

Execution of an expression-statement evaluates the contained expression and then transfers control to the end point of the expression-statement. The end point of an expression-statement is reachable if that expression-statement is reachable.

## Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

selection-statement:  
if-statement  
switch-statement

### The if statement

The if statement selects a statement for execution based on the value of a boolean expression.

if-statement:  
if ( boolean-expression ) embedded-statement  
if ( boolean-expression ) embedded-statement else embedded-statement

An else part is associated with the lexically nearest preceding if that is allowed by the syntax. Thus, an if statement of the form

if (x) if (y) F(); else G();

is equivalent to

if (x) {  
 if (y) {  
 F();  
 }  
 else {  
 G();  
 }  
}

An if statement is executed as follows:

* The boolean-expression (§7.20) is evaluated.
* If the boolean expression yields true, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
* If the boolean expression yields false and if an else part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
* If the boolean expression yields false and if an else part is not present, control is transferred to the end point of the if statement.

The first embedded statement of an if statement is reachable if the if statement is reachable and the boolean expression does not have the constant value false.

The second embedded statement of an if statement, if present, is reachable if the if statement is reachable and the boolean expression does not have the constant value true.

The end point of an if statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an if statement with no else part is reachable if the if statement is reachable and the boolean expression does not have the constant value true.

### 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-sectionsopt }

switch-sections:  
switch-section  
switch-sections switch-section

switch-section:  
switch-labels statement-list

switch-labels:  
switch-label  
switch-labels switch-label

switch-label:  
case constant-expression :  
default :

A switch-statement consists of the keyword switch, followed by a parenthesized expression (called the switch expression), followed by a switch-block. The switch-block consists of zero or more switch-sections, enclosed in braces. Each switch-section consists of one or more switch-labels followed by a statement-list (§8.2.1).

The governing type of a switch statement is established by the switch expression.

* If the type of the switch expression is sbyte, byte, short, ushort, int, uint, long, ulong, bool, char, string, or an enum-type, or if it is the nullable type corresponding to one of these types, then that is the governing type of the switch statement.
* Otherwise, exactly one user-defined implicit conversion (§6.4) must exist from the type of the switch expression to one of the following possible governing types: sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or, a nullable type corresponding to one of those types.
* Otherwise, if no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each case label must denote a value that is implicitly convertible (§6.1) to the governing type of the switch statement. A compile-time error occurs if two or more case labels in the same switch statement specify the same constant value.

There can be at most one default label in a switch statement.

A switch statement is executed as follows:

* The switch expression is evaluated and converted to the governing type.
* If one of the constants specified in a case label in the same switch statement is equal to the value of the switch expression, control is transferred to the statement list following the matched case label.
* If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression, and if a default label is present, control is transferred to the statement list following the default label.
* If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression, and if no default label is present, control is transferred to the end point of the switch statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the “no fall through” rule. The example

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
default:  
 CaseOthers();  
 break;  
}

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to “fall through” to the next switch section, and the example

switch (i) {  
case 0:  
 CaseZero();  
case 1:  
 CaseZeroOrOne();  
default:  
 CaseAny();  
}

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit goto case or goto default statement must be used:

switch (i) {  
case 0:  
 CaseZero();  
 goto case 1;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
default:  
 CaseAny();  
 break;  
}

Multiple labels are permitted in a switch-section. The example

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
case 2:  
default:  
 CaseTwo();  
 break;  
}

is valid. The example does not violate the “no fall through” rule because the labels case 2: and default: are part of the same switch-section.

The “no fall through” rule prevents a common class of bugs that occur in C and C++ when break statements are accidentally omitted. In addition, because of this rule, the switch sections of a switch statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the switch statement above can be reversed without affecting the behavior of the statement:

switch (i) {  
default:  
 CaseAny();  
 break;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
case 0:  
 CaseZero();  
 goto case 1;  
}

The statement list of a switch section typically ends in a break, goto case, or goto default statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a while statement controlled by the boolean expression true is known to never reach its end point. Likewise, a throw or return statement always transfers control elsewhere and never reaches its end point. Thus, the following example is valid:

switch (i) {  
case 0:  
 while (true) F();  
case 1:  
 throw new ArgumentException();  
case 2:  
 return;  
}

The governing type of a switch statement may be the type string. For example:

void DoCommand(string command) {  
 switch (command.ToLower()) {  
 case "run":  
 DoRun();  
 break;  
 case "save":  
 DoSave();  
 break;  
 case "quit":  
 DoQuit();  
 break;  
 default:  
 InvalidCommand(command);  
 break;  
 }  
}

Like the string equality operators (§7.10.7), the switch statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a case label constant.

When the governing type of a switch statement is string, the value null is permitted as a case label constant.

The statement-lists of a switch-block may contain declaration statements (§8.5). The scope of a local variable or constant declared in a switch block is the switch block.

Within a switch block, the meaning of a name used in an expression context must always be the same (§7.6.2.1).

The statement list of a given switch section is reachable if the switch statement is reachable and at least one of the following is true:

* The switch expression is a 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.

## Iteration statements

Iteration statements repeatedly execute an embedded statement.

iteration-statement:  
while-statement  
do-statement  
for-statement  
foreach-statement

### The while statement

The while statement conditionally executes an embedded statement zero or more times.

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

A while statement is executed as follows:

* The boolean-expression (§7.20) is evaluated.
* If the boolean expression yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), control is transferred to the beginning of the while statement.
* If the boolean expression yields false, control is transferred to the end point of the while statement.

Within the embedded statement of a while statement, a break statement (§8.9.1) may be used to transfer control to the end point of the while statement (thus ending iteration of the embedded statement), and a continue statement (§8.9.2) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the while statement).

The embedded statement of a while statement is reachable if the while statement is reachable and the boolean expression does not have the constant value false.

The end point of a while statement is reachable if at least one of the following is true:

* The while statement contains a reachable break statement that exits the while statement.
* The while statement is reachable and the boolean expression does not have the constant value true.

### The do statement

The do statement conditionally executes an embedded statement one or more times.

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

A do statement is executed as follows:

* Control is transferred to the embedded statement.
* When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the boolean-expression (§7.20) is evaluated. If the boolean expression yields true, control is transferred to the beginning of the do statement. Otherwise, control is transferred to the end point of the do statement.

Within the embedded statement of a do statement, a break statement (§8.9.1) may be used to transfer control to the end point of the do statement (thus ending iteration of the embedded statement), and a continue statement (§8.9.2) may be used to transfer control to the end point of the embedded statement.

The embedded statement of a do statement is reachable if the do statement is reachable.

The end point of a do statement is reachable if at least one of the following is true:

* The do statement contains a reachable break statement that exits the do statement.
* The end point of the embedded statement is reachable and the boolean expression does not have the constant value true.

### 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-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

The for-initializer, if present, consists of either a local-variable-declaration (§8.5.1) or a list of statement-expressions (§8.6) separated by commas. The scope of a local variable declared by a for-initializer starts at the local-variable-declarator for the variable and extends to the end of the embedded statement. The scope includes the for-condition and the for-iterator.

The for-condition, if present, must be a boolean-expression (§7.20).

The for-iterator, if present, consists of a list of statement-expressions (§8.6) separated by commas.

A for statement is executed as follows:

* If a for-initializer is present, the variable initializers or statement expressions are executed in the order they are written. This step is 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 (§8.9.1) may be used to transfer control to the end point of the for statement (thus ending iteration of the embedded statement), and a continue statement (§8.9.2) may be used to transfer control to the end point of the embedded statement (thus executing the for-iterator and performing another iteration of the for statement, starting with the for-condition).

The embedded statement of a for statement is reachable if one of the following is true:

* The for statement is reachable and no for-condition is present.
* The for statement is reachable and a for-condition is present and does not have the constant value false.

The end point of a for statement is reachable if at least one of the following is true:

* The for statement contains a reachable break statement that exits the for statement.
* The for statement is reachable and a for-condition is present and does not have the constant value true.

### The foreach statement

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement:  
foreach ( local-variable-type identifier in expression ) embedded-statement

The type and identifier of a foreach statement declare the iteration variable of the statement. If the var identifier is given as the local-variable-type, and no type named var is in scope, the iteration variable is said to be an implicitly typed iteration variable, and its type is taken to be the element type of the foreach statement, as specified below. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a foreach statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (via assignment or the ++ and ‑‑ operators) or pass the iteration variable as a ref or out parameter.

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 (§6.1.8). 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 Ti for which there is an implicit conversion from X to IEnumerable<Ti>, there is a unique type T such that T is not dynamic and for all the other Ti there is an implicit conversion from IEnumerable<T> to IEnumerable<Ti>, 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.
* 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 (§6.2) from T (the element type) to V (the local-variable-type in the foreach statement), an error is produced and no further steps are taken. If x has the value null, a System.NullReferenceException is thrown at run-time.

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 by any anonymous function occurring in the embedded-statement.

For 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 was 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: earlier versions of C# declared v outside of the while loop.)

The body of the finally block is constructed according to the following steps:

* If there is an implicit conversion from E to the System.IDisposable interface, then
* If E is a non-nullable value type then the finally clause is expanded to the semantic equivalent of:

finally {  
 ((System.IDisposable)e).Dispose();  
}

* Otherwise the finally clause is expanded to the semantic equivalent of:

finally {  
 if (e != null) ((System.IDisposable)e).Dispose();  
}

except that if E is a value type, or a type parameter instantiated to a value type, then the cast of e to System.IDisposable will not cause boxing to occur.

* Otherwise, if E is a sealed type, 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.

The following example prints out each value in a two-dimensional array, in element order:

using System;

class Test  
{  
 static void Main() {  
 double[,] values = {  
 {1.2, 2.3, 3.4, 4.5},  
 {5.6, 6.7, 7.8, 8.9}  
 };

foreach (double elementValue in values)  
 Console.Write("{0} ", elementValue);

Console.WriteLine();  
 }  
}

The output produced is as follows:

1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9

In the example

int[] numbers = { 1, 3, 5, 7, 9 };  
foreach (var n in numbers) Console.WriteLine(n);

the type of n is inferred to be int, the element type of numbers.

## Jump statements

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 may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening try statements. In the absence of such try statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening try statements, execution is more complex. If the jump statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.

In the example

using System;

class Test  
{  
 static void Main() {  
 while (true) {  
 try {  
 try {  
 Console.WriteLine("Before break");  
 break;  
 }  
 finally {  
 Console.WriteLine("Innermost finally block");  
 }  
 }  
 finally {  
 Console.WriteLine("Outermost finally block");  
 }  
 }  
 Console.WriteLine("After break");  
 }  
}

the finally blocks associated with two try statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

Before break  
Innermost finally block  
Outermost finally block  
After break

### The break statement

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement.

break-statement:  
break ;

The target of a break statement is the end point of the nearest enclosing switch, while, do, for, or foreach statement. If a break statement is not enclosed by a switch, while, do, for, or foreach statement, a compile-time error occurs.

When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement (§8.9.3) must be used.

A break statement cannot exit a finally block (§8.10). When a break statement occurs within a finally block, the target of the break statement must be within the same finally block; otherwise, a compile-time error occurs.

A break statement is executed as follows:

* If the break statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
* Control is transferred to the target of the break statement.

Because a break statement unconditionally transfers control elsewhere, the end point of a break statement is never reachable.

### The continue statement

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement.

continue-statement:  
continue ;

The target of a continue statement is the end point of the embedded statement of the nearest enclosing while, do, for, or foreach statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs.

When multiple while, do, for, or foreach statements are nested within each other, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement (§8.9.3) must be used.

A continue statement cannot exit a finally block (§8.10). When a continue statement occurs within a finally block, the target of the continue statement must be within the same finally block; otherwise a compile-time error occurs.

A continue statement is executed as follows:

* If the continue statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
* Control is transferred to the target of the continue statement.

Because a continue statement unconditionally transfers control elsewhere, the end point of a continue statement is never reachable.

### The goto statement

The goto statement transfers control to a statement that is marked by a label.

goto-statement:  
goto identifier ;  
goto case constant-expression ;  
goto default ;

The target of a goto identifier statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control out of a nested scope, but not into a nested scope. In the example

using System;

class Test  
{  
 static void Main(string[] args) {  
 string[,] table = {  
 {"Red", "Blue", "Green"},  
 {"Monday", "Wednesday", "Friday"}  
 };

foreach (string str in args) {  
 int row, colm;  
 for (row = 0; row <= 1; ++row)  
 for (colm = 0; colm <= 2; ++colm)  
 if (str == table[row,colm])  
 goto done;

Console.WriteLine("{0} not found", str);  
 continue;  
 done:  
 Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);  
 }  
 }  
}

a goto statement is used to transfer control out of a nested scope.

The target of a goto case statement is the statement list in the immediately enclosing switch statement (§8.7.2), which contains a case label with the given constant value. If the goto case statement is not enclosed by a switch statement, if the constant-expression is not implicitly convertible (§6.1) to the governing type of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant value, a compile-time error occurs.

The target of a goto default statement is the statement list in the immediately enclosing switch statement (§8.7.2), which contains a default label. If the goto default statement is not enclosed by a switch statement, or if the nearest enclosing switch statement does not contain a default label, a compile-time error occurs.

A goto statement cannot exit a finally block (§8.10). When a goto statement occurs within a finally block, the target of the goto statement must be within the same finally block, 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.

### The return statement

The return statement returns control to the current caller of the function in which the return statement appears.

return-statement:  
return expressionopt ;

A return statement with no expression can be used only in a function member that does not compute a value, that is, a method with the result type (§10.6.10) void, the set accessor of a property or indexer, the add and remove accessors of an event, an instance constructor, a static constructor, or a destructor.

A return statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void result type, the get accessor of a property or indexer, or a user-defined operator. An implicit conversion (§6.1) must exist from the type of the expression to the return type of the containing function member.

Return statements can also be used in the body of anonymous function expressions (§7.15), and participate in determining which conversions exist for those functions.

It is a compile-time error for a return statement to appear in a finally block (§8.10).

A return statement is executed as follows:

* If the return statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function 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 (§10.14.1).

Because a return statement unconditionally transfers control elsewhere, the end point of a return statement is never reachable.

### The throw statement

The throw statement throws an exception.

throw-statement:  
throw expressionopt ;

A throw statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type System.Exception, of a class type that derives from System.Exception or of a type parameter type that has System.Exception (or a subclass thereof) as its effective base class. If evaluation of the expression produces null, a System.NullReferenceException is thrown instead.

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 the exception type or a base type of the exception type is considered a match. A general catch clause (§8.10) is considered a match for any exception type. If a matching catch clause is located, the exception propagation is completed by transferring control to the block of that catch clause.
* Otherwise, if the try block or a catch block of S encloses the throw point and if S has a finally block, control is transferred to the finally block. If the finally block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the finally block, processing of the current exception is continued.
* If an exception handler was not located in the current function 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 §10.14.1.
* If the current function is async and void-returning, the synchronization context of the current thread is notified as described in §10.14.2.
* 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.

## The try statement

The try statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the try statement provides the ability to specify a block of code that is always executed when control leaves the try statement.

try-statement:  
try block catch-clauses  
try block finally-clause  
try block catch-clauses finally-clause

catch-clauses:  
specific-catch-clauses general-catch-clauseopt  
specific-catch-clausesopt general-catch-clause

specific-catch-clauses:  
specific-catch-clause  
specific-catch-clauses specific-catch-clause

specific-catch-clause:  
catch ( class-type identifieropt ) block

general-catch-clause:  
catch block

finally-clause:  
finally block

There are three possible forms of try statements:

* A try block followed by one or more catch blocks.
* A try block followed by a finally block.
* A try block followed by one or more catch blocks followed by a finally block.

When a catch clause specifies a class-type, the type must be System.Exception, a type that derives from System.Exception or a type parameter type that has System.Exception (or a subclass thereof) as its effective base class.

When a catch clause specifies both a class-type and an identifier, an exception variable of the given name and type is declared. The exception variable corresponds to a local variable 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 must be the last catch clause.

Some programming languages may support exceptions that are not representable as an object derived from System.Exception, although such exceptions could never be generated by C# code. A general catch clause may be used to catch such exceptions. Thus, a general catch clause is semantically different from one that specifies the type System.Exception, in that the former may also catch exceptions from other languages.

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. Without this restriction, it would be possible to write unreachable catch clauses.

Within a catch block, a throw statement (§8.9.5) with no expression can be used to 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.

In the example

using System;

class Test  
{  
 static void F() {  
 try {  
 G();  
 }  
 catch (Exception e) {  
 Console.WriteLine("Exception in F: " + e.Message);  
 e = new Exception("F");  
 throw; // 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 is would be as follows:

Exception in F: G  
Exception in Main: F

It is a compile-time error for a break, continue, or goto statement to transfer control out of a finally block. When a break, continue, or goto statement occurs in a finally block, the target of the statement must be within the same finally block, 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:
* If the try statement has a finally block, the finally block is executed.
* Control is transferred to the end point of the try statement.
* If an exception is propagated to the try statement during execution of the try block:
* The catch clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located:
* If the matching catch clause declares an exception variable, the exception object is assigned to the exception variable.
* Control is transferred to the matching catch block.
* When and if control reaches the end point of the catch block:
* If the try statement has a finally block, the finally block is executed.
* Control is transferred to the end point of the try statement.
* If an exception is propagated to the try statement during execution of the catch block:
* If the try statement has a finally block, the finally block is executed.
* The exception is propagated to the next enclosing try statement.
* If the try statement has no catch clauses or if no catch clause matches the exception:
* If the try statement has a finally block, the finally block is executed.
* The exception is propagated to the next enclosing try statement.

The statements of a finally block are always executed when control leaves a try statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a break, continue, goto, or return statement, or as a result of propagating an exception out of the try statement.

If an exception is thrown during execution of a finally block, and is not caught within the same finally block, the exception is propagated to the next enclosing try statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the throw statement (§8.9.5).

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.

## 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 (§7.6.12), except that they operate on blocks instead of expressions.

## The lock statement

The lock statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

lock-statement:  
lock ( expression ) embedded-statement

The expression of a lock statement must denote a value of a type known to be a reference-type. No implicit boxing conversion (§6.1.7) is ever performed for the expression of a lock 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.

Locking System.Type objects in order to synchronize access to static data is not recommended. Other code might lock on the same type, which can result in deadlock. A better approach is to synchronize access to static data by locking a private static object. For example:

class Cache  
{  
 private static readonly object synchronizationObject = new object();

public static void Add(object x) {  
 lock (Cache.synchronizationObject) {  
 ...  
 }  
 }

public static void Remove(object x) {  
 lock (Cache.synchronizationObject) {  
 ...  
 }  
 }  
}

## 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 System.IDisposable, which includes a single parameterless method named Dispose. Code that is using a resource can call Dispose to indicate that the resource is no longer needed. If Dispose is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of resource-acquisition is local-variable-declaration then the type of the local-variable-declaration must be either dynamic or a type that can be implicitly converted to System.IDisposable. If the form of resource-acquisition is expression then this expression must be implicitly convertible to System.IDisposable.

Local variables declared in a resource-acquisition are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these local variables (via assignment or the ++ and ‑‑ operators) , take the address of them, or pass them as ref or out parameters.

A using statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a try statement that includes a finally clause. This finally clause disposes of the resource. If a null resource is acquired, then no call to Dispose is made, and no exception is thrown. If the resource is of type dynamic it is dynamically converted through an implicit dynamic conversion (§6.1.8) to IDisposable during acquisition 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, the expansion is

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 ((IDisposable)resource).Dispose();  
 }  
}

Otherwise, when ResourceType is a nullable value type or a reference type other than dynamic, the expansion is

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 if (resource != null) ((IDisposable)resource).Dispose();  
 }  
}

Otherwise, when ResourceType is dynamic, the expansion is

{  
 ResourceType resource = expression;  
 IDisposable d = (IDisposable)resource;  
 try {  
 statement;  
 }  
 finally {  
 if (d != null) d.Dispose();  
 }  
}

In either expansion, the resource variable is read-only in the embedded statement, and the d variable is inaccessible in, and invisible to, the embedded statement.

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

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.

## The yield statement

The yield statement is used in an iterator block (§8.2) to yield a value to the enumerator object (§10.14.4) or enumerable object (§10.14.5) of an iterator or to signal the end of the iteration.

yield-statement:  
yield return expression ;  
yield break ;

yield is not a reserved word; it has special meaning only when used immediately before a return or break keyword. In other contexts, yield can be used as an identifier.

There are several restrictions on where a yield statement can appear, as described in the following.

* It is a compile-time error for a yield statement (of either form) to appear outside a method-body, operator-body or accessor-body
* It is a compile-time error for a yield statement (of either form) to appear inside an anonymous function.
* It is a compile-time error for a yield statement (of either form) to appear in the finally clause of a try statement.
* It is a compile-time error for a yield return statement to appear anywhere in a try statement that contains any catch clauses.

The following example shows some valid and invalid uses of yield statements.

delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {  
 try {  
 yield return 1; // Ok  
 yield break; // Ok  
 }  
 finally {  
 yield return 2; // Error, yield in finally  
 yield break; // Error, yield in finally  
 }

try {  
 yield return 3; // Error, yield return in try...catch  
 yield break; // 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  
}

An implicit conversion (§6.1) must exist from the type of the expression in the yield return statement to the yield type (§10.14.3) of the iterator.

A yield return statement is executed as follows:

* The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the Current property of the enumerator object.
* Execution of the iterator block is suspended. If the yield return statement is within one or more try blocks, the associated finally blocks are not executed at this time.
* The MoveNext method of the enumerator object returns true to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object’s MoveNext method resumes execution of the iterator block from where it was last suspended.

A yield break statement is executed as follows:

* If the yield break statement is enclosed by one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all enclosing try statements have been executed.
* Control is returned to the caller of the iterator block. This is either the MoveNext method or 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.

21

# EXPRESSIONS

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

## Expression classifications

An expression is classified as one of the following:

* A value. Every value has an associated type.
* A variable. Every variable has an associated type, namely the declared type of the variable.
* A namespace. An expression with this classification can only appear as the left hand side of a member-access (§7.6.4). In any other context, an expression classified as a namespace causes a compile-time error.
* A type. An expression with this classification can only appear as the left hand side of a member-access (§7.6.4), or as an operand for the as operator (§7.10.11), the is operator (§7.10.10), or the typeof operator (§7.6.11). In any other context, an expression classified as a type causes a compile-time error.
* A method group, which is a set of overloaded methods resulting from a member lookup (§7.4). A method group may have an associated instance expression and an associated type argument list. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by this (§7.6.7). A method group is permitted in an invocation-expression (§7.6.5) , a delegate-creation-expression (§7.6.10.5) and as the left hand side of an is operator, and can be implicitly converted to a compatible delegate type (§6.6). In any other context, an expression classified as a method group causes a compile-time error.
* A null literal. An expression with this classification can be implicitly converted to a reference type or nullable type.
* An anonymous function. An expression with this classification can be implicitly converted to a compatible delegate type or expression tree type.
* A property access. Every property access has an associated type, namely the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the get or set block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by this (§7.6.7).
* An event access. Every event access has an associated type, namely the type of the event. Furthermore, an event access may have an associated instance expression. An event access may appear as the left hand operand of the += and -= operators (§7.17.3). In any other context, an expression classified as an event access causes a compile-time error.
* An indexer access. Every indexer access has an associated type, namely the element type of the indexer. Furthermore, an indexer access has an associated instance expression and an associated argument list. When an accessor (the get or set block) of an indexer access is invoked, the result of evaluating the instance expression becomes the instance represented by this (§7.6.7), and the result of evaluating the argument list becomes the parameter list of the invocation.
* Nothing. This occurs when the expression is an invocation of a method with a return type of void. An expression classified as nothing is only valid in the context of a statement-expression (§8.6).

The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted above, these categories of expressions are intermediate constructs that are 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 (§7.17.1). Otherwise, the get-accessor is invoked to obtain the current value (§7.1.1).

### Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a value. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

* The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned (§5.3) before its value can be obtained, 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 (§7.5.4) is performed, and the result of the invocation becomes the value of the property access expression.
* The value of an indexer access expression is obtained by invoking the get-accessor of the indexer. If the indexer has no get-accessor, a compile-time error occurs. Otherwise, a function member invocation (§7.5.4) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

## Static and Dynamic Binding

The process of determining the meaning of an operation based on the type or value of constituent expressions (arguments, operands, receivers) is often referred to as binding. For instance the meaning of a method call is determined based on the type of the receiver and arguments. The meaning of an operator is determined based on the type of its operands.

In C# the meaning of an operation is usually determined at compile-time, based on the compile-time type of its constituent expressions. Likewise, if an expression contains an error, the error is detected and reported by the compiler. This approach is known as static binding.

However, if an expression is a dynamic expression (i.e. has the type dynamic) this indicates that any binding that it participates in should be based on its run-time type (i.e. the actual type of the object it denotes at run-time) 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 invocaton: 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 compile-time types of constituent expressions are used in the selection process. However, when one of the constituent expressions in the operations listed above is a dynamic expression, the operation is instead dynamically bound.

### Binding-time

Static binding takes place at compile-time, whereas dynamic binding takes place at run-time. In the following sections, the term binding-time refers to either compile-time or run-time, depending on when the binding takes place.

The following example illustrates the notions of static and dynamic binding and of binding-time:

object o = 5;  
dynamic d = 5;

Console.WriteLine(5); // static binding to Console.WriteLine(int)  
Console.WriteLine(o); // static binding to Console.WriteLine(object)  
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)

The first two calls are statically bound: the overload of Console.WriteLine is picked based on the compile-time type of their argument. Thus, the binding-time is compile-time.

The third call is dynamically bound: the overload of Console.WriteLine is picked based on the 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.

### Dynamic binding

The purpose of dynamic binding is to allow 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 document, 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.*

*The following sections describe for each construct in the language exactly when dynamic binding is applied, what compile time checking – if any – is applied, and what the compile-time result and expression classification is.*

### *Types of constituent expressions*

When an operation is statically bound, the type of a constituent expression (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 constituent expression is determined in different ways depending on the compile-time type of the constituent expression:

* A constituent expression of compile-time type dynamic is considered to have the type of the actual value that the expression evaluates to at runtime
* A constituent expression whose compile-time type is a type parameter is considered to have the type which the type parameter is bound to at runtime
* Otherwise the constituent expression is considered to have its compile-time type.

## Operators

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

There are three kinds of operators:

* Unary operators. The unary operators take one operand and use either prefix notation (such as –x) or postfix notation (such as x++).
* Binary operators. The binary operators take two operands and all use infix notation (such as x + y).
* Ternary operator. Only one ternary operator, ?:, exists; it takes three operands and uses infix notation (c? x: y).

The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators (§7.3.1).

Operands in an expression are evaluated from left to right. For example, in F(i) + G(i++) \* H(i), method F is called using the old value of i, then method G is called with the old value of i, and, finally, method H is called with the new value of i. This is separate from and unrelated to operator precedence.

Certain operators can be overloaded. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (§7.3.2).

### Operator precedence and associativity

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x + y \* z is evaluated as x + (y \* z) because the \* operator has higher precedence than the binary + operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an additive-expression consists of a sequence of multiplicative-expressions separated by + or - operators, thus giving the + and - operators lower precedence than the \*, /, and % operators.

The following table summarizes all operators in order of precedence from highest to lowest:

|  |  |  |
| --- | --- | --- |
| **Section** | **Category** | **Operators** |
| 7.6 | Primary | x.y f(x) a[x] x++ x-- new  typeof default checked unchecked delegate |
| 7.7 | Unary | + - ! ~ ++x --x (T)x |
| 7.8 | Multiplicative | \* / % |
| 7.8 | Additive | + - |
| 7.9 | Shift | << >> |
| 7.10 | Relational and type testing | < > <= >= is as |
| 7.10 | Equality | == != |
| 7.11 | Logical AND | & |
| 7.11 | Logical XOR | ^ |
| 7.11 | Logical OR | | |
| 7.12 | Conditional AND | && |
| 7.12 | Conditional OR | || |
| 7.13 | Null coalescing | ?? |
| 7.14 | Conditional | ?: |
| 7.17, 7.15 | Assignment and lambda expression | = \*= /= %= += -= <<= >>= &= ^= |=  => |

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

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

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

### Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including operator declarations in classes and structs (§10.10). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered, as described in §7.3.3 and §7.3.4.

The overloadable unary operators are:

+ - ! ~ ++ -- true false

Although true and false are not used explicitly in expressions (and therefore are not included in the precedence table in §7.3.1), they are considered operators because they are invoked in several expression contexts: boolean expressions (§7.20) and expressions involving the conditional (§7.14), and conditional logical operators (§7.12).

The overloadable binary operators are:

+ - \* / % & | ^ << >> == != > < >= <=

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the =, &&, ||, ??, ?:, =>, checked, unchecked, new, typeof, default, as, and is operators.

When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded. For example, an overload of operator \* is also an overload of operator \*=. This is described further in §7.17.2. Note that the assignment operator itself (=) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

Cast operations, such as (T)x, are overloaded by providing user-defined conversions (§6.4).

Element access, such as a[x], is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (§10.9).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, op denotes any overloadable unary prefix operator. In the second entry, op denotes the unary postfix ++ and -- operators. In the third entry, op denotes any overloadable binary operator.

|  |  |
| --- | --- |
| **Operator notation** | **Functional notation** |
| op x | operator op(x) |
| x op | operator op(x) |
| x op y | operator op(x, y) |

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the / operator is always a binary operator, always has the precedence level specified in §7.3.1, and is always left-associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of operator == should compare the two operands for equality and return an appropriate bool result.

The descriptions of individual operators in §7.6 through §7.12 specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms unary operator overload resolution, binary operator overload resolution, and numeric promotion, definitions of which are found in the following sections.

### Unary operator overload resolution

An operation of the form op x or x op, where op is an overloadable unary operator, and x is an expression of type X, is processed as follows:

* The set of candidate user-defined operators provided by X for the operation operator op(x) is determined using the rules of §7.3.5.
* If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined unary operator op implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§7.6 and §7.7).
* The overload resolution rules of §7.5.3 are applied to the set of candidate operators to select the best operator with respect to the argument list (x), and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

### Binary operator overload resolution

An operation of the form x op y, where op is an overloadable binary operator, x is an expression of type X, and y is an expression of type Y, is processed as follows:

* The set of candidate user-defined operators provided by X and Y for the operation operator op(x, y) is determined. The set consists of the union of the candidate operators provided by X and the candidate operators provided by Y, each determined using the rules of §7.3.5. If X and Y are the same type, or if X and Y are derived from a common base type, then shared candidate operators only occur in the combined set once.
* If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary operator op implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§7.8 through §7.12). For predefined enum and delegate operators, the only operators considered are those defined by an enum or delegate type that is the binding-time type of one of the operands.
* The overload resolution rules of §7.5.3 are applied to the set of candidate operators to select the best operator with respect to the argument list (x, y), and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

### 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 T0. If T is a nullable type, T0 is its underlying type, otherwise T0 is equal to T.
* For all operator op declarations in T0 and all lifted forms of such operators, if at least one operator is applicable (§7.5.3.1) with respect to the argument list A, then the set of candidate operators consists of all such applicable operators in T0.
* Otherwise, if T0 is object, the set of candidate operators is empty.
* Otherwise, the set of candidate operators provided by T0 is the set of candidate operators provided by the direct base class of T0, or the effective base class of T0 if T0 is a type parameter.

### Numeric promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary \* operator:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);  
float operator \*(float x, float y);  
double operator \*(double x, double y);  
decimal operator \*(decimal x, decimal y);

When overload resolution rules (§7.5.3) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. For example, for the operation b \* s, where b is a byte and s is a short, overload resolution selects operator \*(int, int) as the best operator. Thus, the effect is that b and s are converted to int, and the type of the result is int. Likewise, for the operation i \* d, where i is an int and d is a double, overload resolution selects operator \*(double, double) as the best operator.

#### Unary numeric promotions

Unary numeric promotion occurs for the operands of the predefined +, –, and ~ unary operators. Unary numeric promotion simply consists of converting operands of type sbyte, byte, short, ushort, or char to type int. Additionally, for the unary – operator, unary numeric promotion converts operands of type uint to type long.

#### Binary numeric promotions

Binary numeric promotion occurs for the operands of the predefined +, –, \*, /, %, &, |, ^, ==, !=, >, <, >=, and <= binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the 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 binding-time 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 that the first rule disallows any operations that mix the decimal type with the double and float types. The rule follows from the fact that there are no implicit conversions between the decimal type and the double and float types.

Also note that it is not possible for an operand to be of type ulong when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of ulong as well as the signed integral types.

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

In the example

decimal AddPercent(decimal x, double percent) {  
 return x \* (1.0 + percent / 100.0);  
}

a binding-time error occurs because a decimal cannot be multiplied by a double. The error is resolved by explicitly converting the second operand to decimal, as follows:

decimal AddPercent(decimal x, double percent) {  
 return x \* (decimal)(1.0 + percent / 100.0);  
}

### 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 §7.11.3). Otherwise, the lifted operator unwraps the operands, applies the underlying operator, and wraps the result.

* For the equality operators

== !=

a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is bool. The lifted form is constructed by adding a single ? 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.

## Member lookup

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 (§7.6.2) or a member-access (§7.6.4) in an expression. If the simple-name or member-access occurs as the primary-expression of an invocation-expression (§7.6.5.1), the member is said to be invoked.

If a member is a method or event, or if it is a constant, field or property of either a delegate type (§15) or the type dynamic (§4.7), then the member is said to be invocable*.*

Member lookup considers not only the name of a member but also the number of type parameters the member has and whether the member is accessible. For the purposes of member lookup, generic methods and nested generic types have the number of type parameters indicated in their respective declarations and all other members have zero type parameters.

A member lookup of a name N with K type parameters in a type T is processed as follows:

* First, a set of accessible members named N is determined:
* If T is a type parameter, then the set is the union of the sets of accessible members named N in each of the types specified as a primary constraint or secondary constraint (§10.1.5) for T, along with the set of accessible members named N in object.
* Otherwise, the set consists of all accessible (§3.5) members named N in T, including inherited members and the accessible members named N in object. If T is a constructed type, the set of members is obtained by substituting type arguments as described in §10.3.2. Members that include an override modifier are excluded from the set.
* Next, if K is zero, all nested types whose declarations include type parameters are removed. If K is not zero, all members with a different number of type parameters are removed. Note that when K is zero, methods having type parameters are not removed, since the type inference process (§7.5.2) might be able to infer the type arguments.
* Next, if the member is invoked*, all non-*invocable *members are removed from the set.*
* 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 (§10.1.5). For every member S.M in the set, where S is the type in which the member M is declared, the following rules are applied if S is a class declaration other than object:
* If M is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
* If M is a method, then all 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 §13.2.5.

### Base types

For purposes of member lookup, a type T is considered to have the following base types:

* If T is object, then T has no base type.
* If T is an enum-type, the base types of T are the class types System.Enum, System.ValueType, and object.
* If T is a struct-type, the base types of T are the class types System.ValueType and object.
* If T is a class-type, the base types of T are the base classes of T, including the class type object.
* If T is an interface-type, the base types of T are the base interfaces of T and the class type object.
* 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.

## Function members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

* Methods
* Properties
* Events
* Indexers
* User-defined operators
* Instance constructors
* Static constructors
* Destructors

Except for destructors and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list (§7.5.1) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of generic methods may employ type inference to determine the set of type arguments to pass to the method. This process is described in §7.5.2.

Invocations of methods, indexers, operators and instance constructors employ overload resolution to determine which of a candidate set of function members to invoke. This process is described in §7.5.3.

Once a particular function member has been identified at binding-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in §7.5.4.

The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the 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. |
| T.P | The get accessor of the property P in the class or struct T is invoked. A compile-time error occurs if P is not static or if P is write-only. |
| T.P = value | The set accessor of the property P in the class or struct T is invoked with the argument list (value). A compile-time error occurs if P is not static or if P is read-only. |
| e.P | The get accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e. A binding-time error occurs if P is static or if P is write-only. |
| e.P = value | The set accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e and the argument list (value). A binding-time error occurs if P is static or if P is read-only. |
| 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). |

### Argument lists

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.
* 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 (§10.7), events (§10.8), and user-defined operators (§10.10) are always passed as value parameters (§10.6.1.1). The arguments of indexers (§10.9) are always passed as value parameters (§10.6.1.1) or parameter arrays (§10.6.1.4). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, indexer or delegate invocation are specified as an argument-list:

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt 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 (§10.6.1.1).
* The keyword ref followed by a variable-reference (§5.4), indicating that the argument is passed as a reference parameter (§10.6.1.2). A variable must be definitely assigned (§5.3) before it can be passed as a reference parameter. The keyword out followed by a variable-reference (§5.4), indicating that the argument is passed as an output parameter (§10.6.1.3). A variable is considered definitely assigned (§5.3) following a function member invocation in which the variable is passed as an output parameter.

#### 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 most specific declaration or override of the function member, starting with the static type of the receiver, and searching through its base classes.
* For interface methods and indexers, the parameter list is picked form the most specific definition of the member, starting with the interface type and searching through the base interfaces. If no unique parameter list is found, a parameter list with inaccessible names and no optional parameters is constructed, so that invocations cannot use named parameters or omit optional arguments.
* For partial methods, the parameter list of the defining partial method declaration is used.
* For all other function members and delegates there is only a single parameter list, which is the one used.

The position of an argument or parameter is defined as the number of arguments or parameters preceding it in the argument list or parameter list.

The corresponding parameters for function member arguments are established as follows:

* Arguments in the argument-list of instance constructors, methods, indexers and delegates:
* A positional argument where a fixed parameter occurs at the same position in the parameter list corresponds to that parameter.
* A positional argument of a function member with a parameter array invoked in its normal form corresponds to the parameter array, which must occur at the same position in the parameter list.
* A positional argument of a function member with a parameter array invoked in its expanded form, where no fixed parameter occurs at the same position in the parameter list, corresponds to an element in the parameter array.
* 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.

#### Run-time evaluation of argument lists

During the run-time processing of a function member invocation (§7.5.4), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

* For a value parameter, the argument expression is evaluated and an implicit conversion (§6.1) to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.
* For a reference or output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation. If the variable reference given as a reference or output parameter is an array element of a reference-type, a 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 (§10.6.1.4). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (§7.5.3.1):

* When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression that is implicitly convertible (§6.1) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
* When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more positional arguments for the parameter array, where each argument is an expression that is implicitly convertible (§6.1) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Thus, the example

class Test  
{  
 static void F(int x, int y = -1, int z = -2) {  
 System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);  
 }

static void Main() {  
 int i = 0;  
 F(i++, i++, i++);  
 F(z: i++, x: i++);  
 }  
}

produces the output

x = 0, y = 1, z = 2  
x = 4, y = -1, z = 3

The array co-variance rules (§12.5) permit a value of an array type A[] to be a reference to an instance of an array type B[], provided an implicit reference conversion exists from B to A. Because of these rules, when an array element of a reference-type is passed as a reference or output parameter, a run-time check is required to ensure that the actual element type of the array is identical to that of the parameter. In the example

class Test  
{  
 static void F(ref object x) {...}

static void Main() {  
 object[] a = new object[10];  
 object[] b = new string[10];  
 F(ref a[0]); // 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.

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (§7.6.10.4) was inserted around the expanded parameters. For example, given the declaration

void F(int x, int y, params object[] args);

the following invocations of the expanded form of the method

F(10, 20);  
F(10, 20, 30, 40);  
F(10, 20, 1, "hello", 3.0);

correspond exactly to

F(10, 20, new object[] {});  
F(10, 20, new object[] {30, 40});  
F(10, 20, new object[] {1, "hello", 3.0});

In particular, note that an empty array is created when there are zero arguments given for the parameter array.

When arguments are omitted from a function member with corresponding optional parameters, the default arguments of the function member declaration are implicitly passed. Because these are always constant, their evaluation will not impact the evaluation order of the remaining arguments.

### Type inference

When a generic method is called without specifying type arguments, a type inference process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration:

class Chooser  
{  
 static Random rand = new Random();

public static T Choose<T>(T first, T second) {  
 return (rand.Next(2) == 0)? first: second;  
 }  
}

it is possible to invoke the Choose method without explicitly specifying a type argument:

int i = Chooser.Choose(5, 213); // Calls Choose<int>

string s = Chooser.Choose("foo", "bar"); // Calls Choose<string>

Through type inference, the type arguments int and string are determined from the arguments to the method.

Type inference occurs as part of the binding-time processing of a method invocation (§7.6.5.1) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a binding-time error. However, it often leads to a binding-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

Tr M<X1…Xn>(T1 x1 … Tm xm)

With a method call of the form M(E1 …Em) the task of type inference is to find unique type arguments S1…Sn for each of the type parameters X1…Xn so that the call M<S1…Sn>(E1…Em)becomes valid.

During the process of inference each type parameter Xi is either *fixed* to a particular type Si or *unfixed* with an associated set of *bounds.* Each of the bounds is some type T. Initially each type variable Xi is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

*Note:* Type inference takes place not only when a generic method is called. Type inference for conversion of method groups is described in §7.5.2.13 and finding the best common type of a set of expressions is described in §7.5.2.14.

#### The first phase

For each of the method arguments Ei:

* If Ei is an anonymous function, an explicit parameter type inference (§7.5.2.7) is made from Ei to Ti
* Otherwise, if Ei has a type U and xi is a value parameter then a *lower-bound inference* is made from U to Ti.
* Otherwise, if Ei has a type U and xi is a ref or out parameter then an exact inference is made from U to Ti.
* Otherwise, no inference is made for this argument.

#### The second phase

The second phase proceeds as follows:

* All unfixed type variables Xi which do not depend on (§7.5.2.5) any Xj are fixed (§7.5.2.10).
* If no such type variables exist, all unfixed type variables Xi are fixed for which all of the following hold:
  + There is at least one type variable Xj that depends on Xi
  + Xi 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 Ei with corresponding parameter type Ti where the output types (§7.5.2.4) contain unfixed type variables Xj but the input types (§7.5.2.3) do not, an output type inference (§7.5.2.6) is made from Ei to Ti. Then the second phase is repeated.

#### 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.

#### 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.

#### Dependence

An unfixed type variable Xi *depends directly on* an unfixed type variable Xj if for some argument Ek with type Tk Xj occurs in an input type of Ek with type Tk and Xi occurs in an output type of Ek with type Tk.

Xj *depends on* Xi if Xj depends directly on Xi or if Xi depends directly on Xk and Xk depends on Xj. Thus “depends on” is the transitive but not reflexive closure of “depends directly on”.

#### Output type inferences

An output type inference is made from an expression E to a type T in the following way:

* If E is an anonymous function with inferred return type U (§7.5.2.12) and T is a delegate type or expression tree type with return type Tb, then a lower-bound inference (§7.5.2.9) is made from U to Tb.
* Otherwise, if E is a method group and T is a delegate type or expression tree type with parameter types T1…Tk and return type Tb, and overload resolution of E with the types T1…Tk yields a single method with return type U, then a lower-bound inference is made from U to Tb.
* 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.

#### 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 U1…Uk and T is a delegate type or expression tree type with parameter types V1…Vk then for each Ui an exact inference (§7.5.2.8) is made from Ui to the corresponding Vi.

#### Exact inferences

An exact inference from a type U to a type V is made as follows:

* If V is one of the unfixed Xi then U is added to the set of exact bounds for Xi.
* Otherwise, sets V1…Vk and U1…Uk are determined by checking if any of the following cases apply:
* V is an array type V1[…] and U is an array type U1[…] of the same rank
* V is the type V1? and U is the type U1?
* 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 Ui *to* the corresponding Vi.

* Otherwise no inferences are made.

#### 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 Xi then U is added to the set of lower bounds for Xi.
* Otherwise, if V is the type V1? and U is the type U1? then a lower bound inference is made from U1 to V1.
* Otherwise, sets U1…Uk and V1…Vk are determined by checking if any of the following cases apply:
* V is an array type V1[…]and U is an array type U1[…] (or a type parameter whose effective base type is U1[…]) of the same rank
* V is one of IEnumerable<V1>, ICollection<V1> or IList<V1> and U is a one-dimensional array type U1[](or a type parameter whose effective base type is U1[])
* V is a constructed class, struct, interface or delegate type C<V1…Vk> and there is a unique type C<U1…Uk> 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<U1…Uk>.

(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 Ui *to* the corresponding Vi as follows:

* If Ui is not known to be a reference type then an *exact inference* is made
* Otherwise, if U is an array type then a *lower-bound inference* is made
* Otherwise, if V is C<V1…Vk> 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.

#### 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 Xi then U is added to the set of upper bounds for Xi.
* Otherwise, sets V1…Vk and U1…Uk are determined by checking if any of the following cases apply:
* U is an array type U1[…]and V is an array type V1[…]of the same rank
* U is one of IEnumerable<Ue>, ICollection<Ue> or IList<Ue> and V is a one-dimensional array type Ve[]
* U is the type U1? and V is the type V1?
* 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 Ui *to* the corresponding Vi as follows:

* If Ui is not known to be a reference type then an *exact inference* is made
* Otherwise, if V is an array type then an *upper-bound inference* is made
* Otherwise, if U is C<U1…Uk> then inference depends on the i-th type parameter of C:
* If it is covariant then an *upper-bound inference* is made.
* If it is contravariant then a *lower-bound inference* is made.
* If it is invariant then an *exact inference* is made.
* Otherwise, no inferences are made.

#### Fixing

An unfixed type variable Xi with a set of bounds is fixed as follows:

* The set of *candidate types* Uj starts out as the set of all types in the set of bounds for Xi.
* We then examine each bound for Xi in turn: For each exact bound U of Xi all types Uj which are not identical to U are removed from the candidate set. For each lower bound U of Xi all types Uj to which there is *not* an implicit conversion from U are removed from the candidate set. For each upper bound U of Xi all types Uj from which there is *not* an implicit conversion to U are removed from the candidate set.
* If among the remaining candidate types Uj there is a unique type V from which there is an implicit conversion to all the other candidate types, then Xi is fixed to V.
* Otherwise, type inference fails.

#### 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 result type is determined as follows:

* If the body of F is an expression that has a type, then the inferred result type of F is the type of that expression.
* If the body of F is a block and the set of expressions in the block’s return statements has a best common type T (§7.5.2.14), then the inferred result type of F is T.
* Otherwise, a result 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 (§7.1), 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 result type T, the inferred return type is System.Threading.Tasks.Task<T>.
* If F is non-async and has an inferred result type T, the inferred return type is T.
* Otherwise a return type cannot be inferred for F.

As an example of type inference involving anonymous functions, consider the Select extension method declared in the System.Linq.Enumerable class:

namespace System.Linq  
{  
 public static class Enumerable  
 {  
 public static IEnumerable<TResult> Select<TSource,TResult>(  
 this IEnumerable<TSource> source,  
 Func<TSource,TResult> selector)  
 {  
 foreach (TSource element in source) yield return selector(element);  
 }  
 }  
}

Assuming the System.Linq namespace was imported with a using clause, and given a class Customer with a Name property of type string, the Select method can be used to select the names of a list of customers:

List<Customer> customers = GetCustomerList();  
IEnumerable<string> names = customers.Select(c => c.Name);

The extension method invocation (§7.6.5.2) of Select is processed by rewriting the invocation to a static method invocation:

IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the customers argument is related to the source parameter, inferring T to be Customer. Then, using the anonymous function type inference process described above, c is given type Customer, and the expression c.Name is related to the return type of the selector parameter, inferring S to be string. Thus, the invocation is equivalent to

Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)

and the result is of type IEnumerable<string>.

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.

#### Type inference for conversion of method groups

Similar to calls of generic methods, type inference must also be applied when a method group M containing a generic method is converted to a given delegate type D (§6.6). Given a method

Tr M<X1…Xn>(T1 x1 … Tm xm)

and the method group M being assigned to the delegate type D the task of type inference is to find type arguments S1…Sn so that the expression:

M<S1…Sn>

becomes compatible (§15.1) with D.

Unlike the type inference algorithm for generic method calls, in this case there are only argument types, no argument expressions. In particular, there are no anonymous functions and hence no need for multiple phases of inference.

Instead, all Xi are considered unfixed, and a lower-bound inference is made from each argument type Uj of D to the corresponding parameter type Tj of M. If for any of the Xi no bounds were found, type inference fails. Otherwise, all Xi are fixed to corresponding Si, which are the result of type inference.

#### Finding the best common type of a set of expressions

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous functions with block bodies are found in this way.

Intuitively, given a set of expressions E1…Em this inference should be equivalent to calling a method

Tr M<X>(X x1 … X xm)

with the Ei as arguments.

More precisely, the inference starts out with an unfixed type variable X. Output type inferences are then made from each Ei to X. Finally, X is fixed and, if successful, the resulting type S is the resulting best common type for the expressions. If no such S exists, the expressions have no best common type.

### Overload resolution

Overload resolution is a binding-time mechanism for selecting the best function member to invoke given an argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

* Invocation of a method named in an invocation-expression (§7.6.5.1).
* Invocation of an instance constructor named in an object-creation-expression (§7.6.10.1).
* Invocation of an indexer accessor through an element-access (§7.6.6).
* Invocation of a predefined or user-defined operator referenced in an expression (§7.3.3 and §7.3.4).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a method invocation does not include methods marked override (§7.4), and methods in a base class are not candidates if any method in a derived class is applicable (§7.6.5.1).

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

* Given the set of applicable candidate function members, the best function member in that set is located. If the set contains only one function member, then that function member is the best function member. Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in §7.5.3.2. If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a binding-time error occurs.

The following sections define the exact meanings of the terms applicable function member and better function member.

#### 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 §7.5.1.1, and any parameter to which no argument corresponds is an optional parameter.
* For each argument in A, the parameter passing mode of the argument (i.e., value, ref, or out) is identical to the parameter passing mode of the corresponding parameter, and
* for a value parameter or a parameter array, an implicit conversion (§6.1) exists from the argument to the type of the corresponding parameter, or
* for a ref or out parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a ref or out parameter is an alias for the argument passed.

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 may instead be applicable in its expanded form:

* The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list A matches the total number of parameters. If A has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
* Otherwise, the expanded form is applicable if for each argument in A the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
* for a fixed value parameter or a value parameter created by the expansion, an implicit conversion (§6.1) exists from the type of the argument to the type of the corresponding parameter, or
* for a ref or out parameter, the type of the argument is identical to the type of the corresponding parameter.

#### 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 { E1, E2, ..., EN } and two applicable function members MP and MQ with parameter types { P1, P2, ..., PN } and { Q1, Q2, ..., QN }, MP is defined to be a better function member than MQ if

* for each argument, the implicit conversion from EX to QX is not better than the implicit conversion from EX to PX, and
* for at least one argument, the conversion from EX to PX is better than the conversion from EX to QX.

When performing this evaluation, if MP or MQ is applicable in its expanded form, then PX or QX refers to a parameter in the expanded form of the parameter list.

In case the parameter type sequences {P1, P2, …, PN} and {Q1, Q2, …, QN} are equivalent (i.e. each Pi has an identity conversion to the corresponding Qi), the following tie-breaking rules are applied, in order, to determine the better function member.

* If MP is a non-generic method and MQ is a generic method, then MP is better than MQ.
* Otherwise, if MP is applicable in its normal form and MQ has a params array and is applicable only in its expanded form, then MP is better than MQ.
* Otherwise, if MP has more declared parameters than MQ, then MP is better than MQ. This can occur if both methods have params arrays and are applicable only in their expanded forms.
* Otherwise if all parameters of MP have a corresponding argument whereas default arguments need to be substituted for at least one optional parameter in MQ then MP is better than MQ.
* Otherwise, if MP has more specific parameter types than MQ, then MP is better than MQ. Let {R1, R2, …, RN} and {S1, S2, …, SN} represent the uninstantiated and unexpanded parameter types of MP and MQ. MP’s parameter types are more specific than MQ’s if, for each parameter, RX is not less specific than SX, and, for at least one parameter, RX is more specific than SX:
* 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.
* Otherwise, neither function member is better.

#### Better conversion from expression

Given an implicit conversion C1 that converts from an expression E to a type T1, and an implicit conversion C2 that converts from an expression E to a type T2, C1 is a better conversion than C2 if at least one of the following holds:

* E has a type S and an identity conversion exists from S to T1 but not from S to T2
* E is not an anonymous function and T1 is a better conversion target than T2 (§7.5.3.5)
* E is an anonymous function, T1 is either a delegate type D1 or an expression tree type Expression<D1>, T2 is either a delegate type D2 or an expression tree type Expression<D2> and one of the following holds:
* D1 is a better conversion target than D2
* D1 and D2 have identical parameter lists, and one of the following holds:
* D1 has a return type Y1, and D2 has a return type Y2, an inferred return type X exists for E in the context of that parameter list (§7.5.2.12), and the conversion from X to Y1 is better than the conversion from X to Y2
* E is async, D1 has a return type Task<Y1>, and D2 has a return type Task<Y2>, an inferred return type Task<X> exists for E in the context of that parameter list (§7.5.2.12), and the conversion from X to Y1 is better than the conversion from X to Y2
* D1 has a return type Y, and D2 is void returning

#### Better conversion from type

Given a conversion C1 that converts from a type S to a type T1, and a conversion C2 that converts from a type S to a type T2, C1 is a better conversion than C2 if at least one of the following holds:

* An identity conversion exists from S to T1 but not from S to T2
* T1 is a better conversion target than T2 (§7.5.3.5)

#### Better conversion target

Given two different types T1 and T2, T1 is a better conversion target than T2 if at least one of the following holds:

* An implicit conversion from T1 to T2 exists, and no implicit conversion from T2 to T1 exists
* T1 is a signed integral type and T2 is an unsigned integral type. Specifically:
* T1 is sbyte and T2 is byte, ushort, uint, or ulong
* T1 is short and T2 is ushort, uint, or ulong
* T1 is int and T2 is uint, or ulong
* T1 is long and T2 is ulong

#### Overloading in generic classes

While signatures as declared must be unique, it is possible that substitution of type arguments results in identical signatures. In such cases, the tie-breaking rules of overload resolution above will pick the most specific member.

The following examples show overloads that are valid and invalid according to this rule:

interface I1<T> {...}

interface I2<T> {...}

class G1<U>  
{  
 int F1(U u); // Overload resulotion for G<int>.F1  
 int F1(int i); // will pick non-generic

void F2(I1<U> a); // Valid overload  
 void F2(I2<U> a);  
}

class G2<U,V>  
{  
 void F3(U u, V v); // Valid, but overload resolution for  
 void F3(V v, U u); // G2<int,int>.F3 will fail

void F4(U u, I1<V> v); // Valid, but overload resolution for   
 void F4(I1<V> v, U u); // G2<I1<int>,int>.F4 will fail

void F5(U u1, I1<V> v2); // Valid overload  
 void F5(V v1, U u2);

void F6(ref U u); // valid overload  
 void F6(out V v);  
}

### Compile-time checking of dynamic overload resolution

For most dynamically bound operations the set of possible candidates for resolution is unknown at compile-time. In certain cases, however the candidate set is known at compile-time:

* Static method calls with dynamic arguments
* Instance method calls where the receiver is not a dynamic expression
* Indexer calls where the receiver is not a dynamic expression
* Constructor calls with dynamic arguments

In these cases a limited compile-time check is performed for each candidate to see if any of them could possibly apply at run-time.This check consists of the following steps:

* Partial type inference: Any type argument that does not depend directly or indirectly on an argument of type dynamic is inferred using the rules of §7.5.2. The remaining type arguments are unknown.
* Partial applicability check: Applicability is checked according to §7.5.3.1, but ignoring parameters whose types are unknown.

If no candidate passes this test, a compile-time error occurs.

### Function member invocation

This section 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 instance constructors, static methods, static property accessors, and user-defined operators. Static function members are always non-virtual.
* Instance function members. These are instance methods, 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 (§7.6.7).

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 §7.5.1.
* M is invoked.
* If M is an instance function member declared in a value-type:
* E is evaluated. If this evaluation causes an exception, then no further steps are executed.
* If E is not classified as a variable, then a temporary local variable of E’s type is created and the value of E is assigned to that variable. E is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as this within M, 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 §7.5.1.
* M is invoked. The variable referenced by E becomes the variable referenced by this.
* If M is an instance function member declared in a reference-type:
* E is evaluated. If this evaluation causes an exception, then no further steps are executed.
* The argument list is evaluated as described in §7.5.1.
* If the type of E is a value-type, a boxing conversion (§4.3.1) is performed to convert E to type object, and E is considered to be of type object in the following steps. In this case, M could only be a member of System.Object.
* The value of E is checked to be valid. If the value of E is null, a System.NullReferenceException is thrown and no further steps are executed.
* The function member implementation to invoke is determined:
* If the binding-time type of E is an interface, the function member to invoke is the implementation of M provided by the run-time type of the instance referenced by E. This function member is determined by applying the interface mapping rules (§13.4.4) to determine the implementation of M provided by the 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 (§10.6.3) 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.

#### Invocations on boxed instances

A function member implemented in a value-type can be invoked through a boxed instance of that value-type in the following situations:

* When the function member is an override of a method inherited from type object and is invoked through an instance expression of type object.
* When the function member is an implementation of an interface function member and is invoked through an instance expression of an interface-type.
* When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the value-type, and this variable becomes the variable referenced by this within the function member invocation. In particular, 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.

## Primary expressions

Primary expressions include the simplest forms of expressions.

primary-expression:   
primary-no-array-creation-expression  
array-creation-expression

primary-no-array-creation-expression:  
literal  
simple-name  
parenthesized-expression  
member-access  
invocation-expression  
element-access  
this-access  
base-access  
post-increment-expression  
post-decrement-expression  
object-creation-expression  
delegate-creation-expression  
anonymous-object-creation-expression  
typeof-expression  
 checked-expression  
unchecked-expression   
default-value-expression  
anonymous-method-expression

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];

### Literals

A primary-expression that consists of a literal (§2.4.4) is classified as a value.

### Simple names

A simple-name consists of an identifier, optionally followed by a type argument list:

simple-name:  
identifier type-argument-listopt

A simple-name is either of the form I or of the form I<A1, ..., AK>, where I is a single identifier and <A1, ..., AK> is an optional type-argument-list. When no type-argument-list is specified, consider K to be zero. The simple-name is evaluated and classified as follows:

* If K is zero and the simple-name appears within a block and if the block’s (or an enclosing block’s) local variable declaration space (§3.3) contains a local variable, parameter or constant with name I, then the simple-name refers to that local variable, parameter or constant and is classified as a variable or value.
* If K is zero and the simple-name appears within the body of a generic method declaration and if that declaration includes a type parameter with name I, then the simple-name refers to that type parameter.
* Otherwise, for each instance type T (§10.3.1), starting with the instance type of the immediately enclosing type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
* If K is zero and the declaration of T includes a type parameter with name I, then the simple-name refers to that type parameter.
* Otherwise, if a member lookup (§7.4) of I in T with K type arguments produces a match:
* If T is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of this. If a type argument list was specified, it is used in calling a generic method (§7.6.5.1).
* Otherwise, if T is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the block of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (§7.6.4) of the form this.I. This can only happen when K is zero.
* Otherwise, the result is the same as a member access (§7.6.4) of the form T.I or T.I<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 an error occurs.

Note that this entire step is exactly parallel to the corresponding step in the processing of a namespace-or-type-name (§3.8).

* Otherwise, the simple-name is undefined and a compile-time error occurs.

#### 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 (§3.3) immediately enclosing that occurrence, every other occurrence of the same identifier as a full simple-name in an expression or declarator must refer to the same entity. This rule ensures that the meaning of a name is always the same within a given block, switch block, for-, foreach- or using-statement, or anonymous function.

The example

class Test  
{  
 double x;

void F(bool b) {  
 x = 1.0;  
 if (b) {  
 int x;  
 x = 1;  
 }  
 }  
}

results in a compile-time error because x refers to different entities within the outer block (the extent of which includes the nested block in the if statement). In contrast, the example

class Test  
{  
 double x;

void F(bool b) {  
 if (b) {  
 x = 1.0;  
 }  
 else {  
 int x;  
 x = 1;  
 }  
 }  
}

is permitted because the name x is never used in the outer block.

Note that the rule of invariant meaning applies only to simple names. It is perfectly valid for the same identifier to have one meaning as a simple name and another meaning as right operand of a member access (§7.6.4). For example:

struct Point  
{  
 int x, y;

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

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.

### 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.

### Member access

A member-access consists of a primary-expression, a predefined-type, or a qualified-alias-member, followed by a “.” token, followed by an identifier, optionally followed by a type-argument-list.

member-access:  
primary-expression . identifier type-argument-listopt  
predefined-type . identifier type-argument-listopt  
qualified-alias-member . identifier type-argument-listopt

predefined-type: one of  
bool byte char decimal double float int long  
object sbyte short string uint ulong ushort

The qualified-alias-member production is defined in §9.7.

A member-access is either of the form E.I or of the form E.I<A1, ..., AK>, where E is a primary-expression, I is a single identifier and <A1, ..., AK> is an optional type-argument-list. When no type-argument-list is specified, consider K to be zero.

A member-access with a primary-expression of type dynamic is dynamically bound (§7.2.2). In this case the compiler classifies the member access as a property access of type dynamic. The rules below to determine the meaning of the member-access are then applied at 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 must 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 a predefined-type or a primary-expression classified as a type, if E is not a type parameter, and if a member lookup (§7.4) of I in E with K type parameters produces a match, then E.I is evaluated and classified as follows:
* If I identifies a type, then the result is that type constructed with the given type arguments.
* If I identifies one or more methods, then the result is a method group with no associated instance expression. If a type argument list was specified, it is used in calling a generic method (§7.6.5.1).
* If I identifies a static property, then the result is a property access with no associated instance expression.
* If I identifies a static field:
* If the field is readonly and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field I in E.
* Otherwise, the result is a variable, namely the static field I in E.
* If I identifies a static event:
* If the reference occurs within the class or struct in which the event is declared, and the event was declared without event-accessor-declarations (§10.8), then E.I is processed exactly as if I were a static field.
* Otherwise, the result is an event access with no associated instance expression.
* If I identifies a constant, then the result is a value, namely the value of that constant.
* If I identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
* Otherwise, E.I is an invalid member reference, and a compile-time error occurs.
* If E is a property access, indexer access, variable, or value, the type of which is T, and a member lookup (§7.4) of I in T with K type arguments produces a match, then E.I is evaluated and classified as follows:
* First, if E is a property or indexer access, then the value of the property or indexer access is obtained (§7.1.1) and E is reclassified as a value.
* If I identifies one or more methods, then the result is a method group with an associated instance expression of E. If a type argument list was specified, it is used in calling a generic method (§7.6.5.1).
* If I identifies an instance property, then the result is a property access with an associated instance expression of E.
* If T is a class-type and I identifies an instance field of that class-type:
* If the value of E is null, then a System.NullReferenceException is thrown.
* Otherwise, if the field is readonly and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field I in the object referenced by E.
* Otherwise, the result is a variable, namely the field I in the object referenced by E.
* If T is a struct-type and I identifies an instance field of that struct-type:
* If E is a value, or if the field is readonly and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field I in the struct instance given by E.
* Otherwise, the result is a variable, namely the field I in the struct instance given by E.
* If I identifies an instance event:
* If the reference occurs within the class or struct in which the event is declared, and the event was declared without event-accessor-declarations (§10.8), and the reference does not occur as the left-hand side of a += or -= operator, then E.I is processed exactly as if I was an instance field.
* Otherwise, the result is an event access with an associated instance expression of E.
* Otherwise, an attempt is made to process E.I as an extension method invocation (§7.6.5.2). If this fails, E.I is an invalid member reference, and a binding-time error occurs.

#### Identical simple names and type names

In a member access of the form E.I, if E is a single identifier, and if the meaning of E as a simple-name (§7.6.2) is a constant, field, property, local variable, or parameter with the same type as the meaning of E as a type-name (§3.8), then both possible meanings of E are permitted. The two possible meanings of E.I are never ambiguous, since I must necessarily be a member of the type E in both cases. In other words, the rule simply permits access to the static members and nested types of E where a compile-time error would otherwise have occurred. For example:

struct Color  
{  
 public static readonly Color White = new Color(...);  
 public static readonly Color Black = new Color(...);

public Color Complement() {...}  
}

class A  
{  
 public Color Color; // Field Color of type Color

void F() {  
 Color = Color.Black; // References Color.Black static member  
 Color = Color.Complement(); // Invokes Complement() on Color field  
 }

static void G() {  
 Color c = Color.White; // References Color.White static member  
 }  
}

Within the A class, those occurrences of the Color identifier that reference the Color type are underlined, and those that reference the Color field are not underlined.

#### Grammar ambiguities

The productions for simple-name (§7.6.2) and member-access (§7.6.4) can give rise to ambiguities in the grammar for expressions. For example, the statement:

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

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

If a sequence of tokens can be parsed (in context) as a simple-name (§7.6.2), member-access (§7.6.4), or pointer-member-access (§18.5.2) ending with a type-argument-list (§4.4.1), the token immediately following the closing > token is examined. If it is one of

( ) ] } : ; , . ? == != | ^

then the type-argument-list is retained as part of the simple-name, member-access or pointer-member-access and any other possible parse of the sequence of tokens is discarded. Otherwise, the type-argument-list is not considered to be part of the simple-name, member-access or pointer-member-access, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a type-argument-list in a namespace-or-type-name (§3.8). The statement

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

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

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

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

### Invocation expressions

An invocation-expression is used to invoke a method.

invocation-expression:  
primary-expression ( argument-listopt )

An invocation-expression is dynamically bound (§7.2.2) if at least one of the following holds:

* The primary-expression has compile-time type dynamic.
* At least one argument of the optional argument-list has compile-time type dynamic and the primary-expression does not have a delegate type.

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 which have the compile-time type dynamic. If the primary-expression does not have compile-time type dynamic, then the method invocation undergoes a limited compile time check as described in §7.5.4.

The primary-expression of an invocation-expression must be a method group or a value of a delegate-type. If the primary-expression is a method group, the invocation-expression is a method invocation (§7.6.5.1). If the primary-expression is a value of a delegate-type, the invocation-expression is a delegate invocation (§7.6.5.3). If the primary-expression is neither a method group nor a value of a delegate-type, a binding-time error occurs.

The optional argument-list (§7.5.1) provides values or variable references for the parameters of the method.

The result of evaluating an invocation-expression is classified as follows:

* If the invocation-expression invokes a method or delegate that returns void, the result is nothing. An expression that is classified as nothing is permitted only in the context of a statement-expression (§8.6) or as the body of a lambda-expression (§7.15). Otherwise a binding-time error occurs.
* Otherwise, the result is a value of the type returned by the method or delegate.

#### Method invocations

For a method invocation, the primary-expression of the invocation-expression must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the argument-list.

The binding-time processing of a method invocation of the form M(A), where M is a method group (possibly including a type-argument-list), and A is an optional argument-list, consists of the following steps:

* The set of candidate methods for the method invocation is constructed. For each method F associated with the method group M:
* If F is non-generic, F is a candidate when:
* M has no type argument list, and
* F is applicable with respect to A (§7.5.3.1).
* If F is generic and M has no type argument list, F is a candidate when:
* Type inference (§7.5.2) succeeds, inferring a list of type arguments for the call, and
* Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (§4.4.4), and the parameter list of F is applicable with respect to A (§7.5.3.1).
* If F is generic and M includes a type argument list, F is a candidate when:
* F has the same number of method type parameters as were supplied in the type argument list, and
* Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (§4.4.4), and the parameter list of F is applicable with respect to A (§7.5.3.1).
* The set of candidate methods is reduced to contain only methods from the most derived types: For each method C.F in the set, where C is the type in which the method F is declared, all methods declared in a base type of C are removed from the set. Furthermore, if C is a class type other than object, all methods declared in an interface type are removed from the set. (This latter rule only has affect 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.)
* If the resulting set of candidate methods is empty, then further processing along the following steps are abandoned, and instead an attempt is made to process the invocation as an extension method invocation (§7.6.5.2). If this fails, then no applicable methods exist, and a binding-time error occurs.
* The best method of the set of candidate methods is identified using the overload resolution rules of §7.5.3. If a single best method cannot be identified, the method invocation is ambiguous, and a binding-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
* Final validation of the chosen best method is performed:
* The method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a simple-name or a member-access through a type. If the best method is an instance method, the method group must have resulted from a simple-name, a member-access through a variable or value, or a base-access. If neither of these requirements is true, a binding-time error occurs.
* If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (§4.4.4) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a binding-time error occurs.

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 §7.5.4.

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.

#### Extension method invocations

In a method invocation (§7.5.5.1) of one of the forms

expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )

if the normal processing of the invocation finds no applicable methods, an attempt is made to process the construct as an extension method invocation. If expr or any of the args has compile-time type dynamic, extension methods will not apply.

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:

* Ci is a non-generic, non-nested class
* The name of Mj is identifier
* Mj 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 Mj.

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 Ci with eligible extension methods Mj, 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 Ci with eligible extension methods Mj, then the set of those extension methods is the candidate set.
* If no candidate set is found in any enclosing namespace declaration or compilation unit, a compile-time error occurs.
* Otherwise, overload resolution is applied to the candidate set as described in (§7.5.3). If no single best method is found, a compile-time error occurs.
* C is the type within which the best method is declared as an extension method.

Using C as a target, the method call is then processed as a static method invocation (§7.5.4).

The preceding rules mean that instance methods take precedence over extension methods, that extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations, and that extension methods declared directly in a namespace take precedence over extension methods imported into that same namespace with a using namespace directive. For example:

public static class E  
{  
 public static void F(this object obj, int i) { }

public static void F(this object obj, string s) { }  
}

class A { }

class B  
{  
 public void F(int i) { }  
}

class C  
{  
 public void F(object obj) { }  
}

class X  
{  
 static void Test(A a, B b, C c) {  
 a.F(1); // E.F(object, int)  
 a.F("hello"); // E.F(object, string)

b.F(1); // B.F(int)  
 b.F("hello"); // E.F(object, string)

c.F(1); // C.F(object)  
 c.F("hello"); // C.F(object)  
 }  
}

In the example, B’s method takes precedence over the first extension method, and C’s method takes precedence over both extension methods.

public static class C  
{  
 public static void F(this int i) { Console.WriteLine("C.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("C.G({0})", i); }  
 public static void H(this int i) { Console.WriteLine("C.H({0})", i); }  
}

namespace N1  
{  
 public static class D  
 {  
 public static void F(this int i) { Console.WriteLine("D.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("D.G({0})", i); }  
 }  
}

namespace N2  
{  
 using N1;

public static class E  
 {  
 public static void F(this int i) { Console.WriteLine("E.F({0})", i); }  
 }

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.

#### Delegate invocations

For a delegate invocation, the primary-expression of the invocation-expression must be a value of a delegate-type. Furthermore, considering the delegate-type to be a function member with the same parameter list as the delegate-type, the delegate-type must be applicable (§7.5.3.1) with respect to the argument-list of the invocation-expression.

The 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 value of D is checked to be valid. If the value of D is null, a System.NullReferenceException is thrown and no further steps are executed.
* Otherwise, D is a reference to a delegate instance. Function member invocations (§7.5.4) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

### Element access

An element-access consists of a primary-no-array-creation-expression, followed by a “[“ token, followed by an argument-list, followed by a “]” token. The argument-list consists of one or more 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 (§7.2.2) if at least one of the following holds:

* The primary-no-array-creation-expression has compile-time type dynamic.
* At least one expression of the argument-list has compile-time type dynamic and the primary-no-array-creation-expression does not have an array type.

In this case the compiler classifies the element-access as a value of type dynamic. The rules below to determine the meaning of the element-access are then applied at 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 §7.5.4.

If the primary-no-array-creation-expression of an element-access is a value of an array-type, the element-access is an array access (§7.6.6.1). Otherwise, the primary-no-array-creation-expression must be a variable or value of a class, struct, or interface type that has one or more indexer members, in which case the element-access is an indexer access (§7.6.6.2).

#### Array access

For an array access, the primary-no-array-creation-expression of the element-access must be a value of an array-type. Furthermore, the argument-list of an array access is not allowed to contain named arguments.The number of expressions in the argument-list must be the same as the rank of the array-type, and each expression must be of type int, uint, long, ulong, or must be implicitly convertible to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value(s) of the expression(s) in the argument-list.

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 (§6.1) to one of the following types is performed: int, uint, long, ulong. The first type in this list for which an implicit conversion exists is chosen. 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.

#### Indexer access

For an indexer access, the primary-no-array-creation-expression of the element-access must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the argument-list of the element-access.

The binding-time processing of an indexer access of the form P[A], where P is a primary-no-array-creation-expression of a class, struct, or interface type T, and A is an argument-list, consists of the following steps:

* The set of indexers provided by T is constructed. The set consists of all indexers declared in T or a base type of T that are not override declarations and are accessible in the current context (§3.5).
* The set is reduced to those indexers that are applicable and not hidden by other indexers. The following rules are applied to each indexer S.I in the set, where S is the type in which the indexer I is declared:
* If I is not applicable with respect to A (§7.5.3.1), then I is removed from the set.
* If I is applicable with respect to A (§7.5.3.1), then all indexers declared in a base type of S are removed from the set.
* If I is applicable with respect to A (§7.5.3.1) and S is a class type other than object, all indexers declared in an interface are removed from the set.
* If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a binding-time error occurs.
* The best indexer of the set of candidate indexers is identified using the overload resolution rules of §7.5.3. If a single best indexer cannot be identified, the indexer access is ambiguous, and a binding-time error occurs.
* The index expressions of the argument-list are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of P and an associated argument list of A.

Depending on the context in which it is used, an indexer access causes invocation of either the get-accessor or the set-accessor of the indexer. If the indexer access is the target of an assignment, the set-accessor is invoked to assign a new value (§7.17.1). In all other cases, the get-accessor is invoked to obtain the current value (§7.1.1).

### This access

A this-access consists of the reserved word this.

this-access:  
this

A this-access is permitted only in the block of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

* When this is used in a primary-expression within an instance constructor of a class, it is classified as a value. The type of the value is the instance type (§10.3.1) of the class within which the usage occurs, and the value is a reference to the object being constructed.
* When this is used in a primary-expression within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the instance type (§10.3.1) of the class within which the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.
* When this is used in a primary-expression within an instance constructor of a struct, it is classified as a variable. The type of the variable is the instance type (§10.3.1) of the struct within which the usage occurs, and the variable represents the struct being constructed. The this variable of an instance constructor of a struct behaves exactly the same as an out parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
* When this is used in a primary-expression within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the instance type (§10.3.1) of the struct within which the usage occurs.
* If the method or accessor is not an iterator (§10.14), the this variable represents the struct for which the method or accessor was invoked, and behaves exactly the same as a ref parameter of the struct type.
* If the method or accessor is an iterator, the this variable represents a copy of the struct for which the method or accessor was invoked, and behaves exactly the same as a value parameter of the struct type.

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.

### Base access

A base-access consists of the reserved word base followed by either a “.” token and an identifier or an argument-list enclosed in square brackets:

base-access:  
base . identifier  
base [ argument-list ]

A base-access is used to access base class members that are hidden by similarly named members in the current class or struct. A base-access is permitted only in the block of an instance constructor, an instance method, or an instance accessor. When base.I occurs in a class or struct, I must denote a member of the base class of that class or struct. Likewise, when base[E] occurs in a class, an applicable indexer must exist in the base class.

At binding-time, base-access expressions of the form base.I and base[E] are evaluated exactly as if they were written ((B)this).I and ((B)this)[E], where B is the base class of the class or struct in which the construct occurs. Thus, base.I and base[E] correspond to this.I and this[E], except this is viewed as an instance of the base class.

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 (§7.5.4) is changed. The function member that is invoked is determined by finding the most derived implementation (§10.6.3) of the function member with respect to B (instead of with respect to the 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.

### Postfix increment and decrement operators

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression --

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the primary-expression has the compile-time type dynamic then the operator is dynamically bound (§7.2.2), the post-increment-expression or post-decrement-expression has the compile-time type dynamic and the following rules are applied at 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 must have both a get and a set accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. Predefined ++ and -- operators exist for the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any enum type. The predefined ++ operators return the value produced by adding 1 to the operand, and the predefined -- operators return the value produced by subtracting 1 from the operand. In a checked context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a System.OverflowException is thrown.

The 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 selected operator is invoked with the saved value of x as its argument.
* The value returned by the operator is stored in the location given by the evaluation of x.
* The saved value of x becomes the result of the operation.
* If x is classified as a property or indexer access:
* The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent get and set accessor invocations.
* The get accessor of x is invoked and the returned value is saved.
* The selected operator is invoked with the saved value of x as its argument.
* The set accessor of x is invoked with the value returned by the operator as its value argument.
* The saved value of x becomes the result of the operation.

The ++ and -- operators also support prefix notation (§7.7.5). Typically, the result of x++ or x-- is the value of x before the operation, whereas the result of ++x or --x is the value of x after the operation. In either case, x itself has the same value after the operation.

An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

### The new operator

The new operator is used to create new instances of types.

There are three forms of new expressions:

* Object creation expressions are used to create new instances of class types and value types.
* Array creation expressions are used to create new instances of array types.
* Delegate creation expressions are used to create new instances of delegate types.

The new operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when new is used to create instances of value types.

#### 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-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

object-or-collection-initializer:  
object-initializer  
collection-initializer

The type of an object-creation-expression must be a class-type, a value-type or a type-parameter. The type cannot be an abstract class-type.

The optional argument-list (§7.5.1) is permitted only if the type is a class-type or a struct-type.

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Processing of an object creation expression that includes an object initializer or collection initializer consists of first processing the instance constructor and then processing the member or element initializations specified by the object initializer (§7.6.10.2) or collection initializer (§7.6.10.3).

If any of the arguments in the optional *argument-list* has the compile-time type dynamic then the object-creation-expression is dynamically bound (§7.2.2) and the following rules are applied at 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 §7.5.4.

The binding-time processing of an object-creation-expression of the form new T(A), where T is a class-type or a value-type and A is an optional argument-list, consists of the following steps:

* If T is a value-type and A is not present:
* The object-creation-expression is a default constructor invocation. The result of the object-creation-expression is a value of type T, namely the default value for T as defined in §4.1.1.
* Otherwise, if T is a type-parameter and A is not present:
* If no value type constraint or constructor constraint (§10.1.5) has been specified for T, a binding-time error occurs.
* The result of the object-creation-expression is a value of the 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 class-type, a compile-time error occurs.
* The instance constructor to invoke is determined using the overload resolution rules of §7.5.3. The set of candidate instance constructors consists of all accessible instance constructors declared in T which are applicable with respect to A (§7.5.3.1). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a binding-time error occurs.
* The result of the object-creation-expression is a value of type T, namely the value produced by invoking the instance constructor determined in the step above.
* Otherwise, the object-creation-expression is invalid, and a binding-time error occurs.

Even if the object-creation-expression is dynamically bound, the compile-time type is still T.

The 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 (§5.2).
* The instance constructor is invoked according to the rules of function member invocation (§7.5.4). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as this.
* If T is a struct-type:
* An instance of type T is created by allocating a temporary local variable. Since an instance constructor of a struct-type is required to definitely assign a value to each field of the instance being created, no initialization of the temporary variable is necessary.
* The instance constructor is invoked according to the rules of function member invocation (§7.5.4). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as this.

#### Object initializers

An object initializer specifies values for zero or more fields or properties of an object.

object-initializer:  
{ member-initializer-listopt }  
{ member-initializer-list , }

member-initializer-list:  
member-initializer  
member-initializer-list , member-initializer

member-initializer:  
identifier = initializer-value

initializer-value:  
expression  
object-or-collection-initializer

An object initializer consists of a sequence of member initializers, enclosed by { and } tokens and separated by commas. Each member initializer must name an accessible field or property of the object being initialized, followed by an equals sign and an expression 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 (§7.17.1) to the field or property.

A member initializer that specifies an object initializer after the equals sign is a nested object initializer, 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 must be of a collection type that satisfies the requirements specified in §7.6.10.3.

The following class represents a point with two coordinates:

public class Point  
{  
 int x, y;

public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

An instance of Point can be created and initialized as follows:

Point a = new Point { X = 0, Y = 1 };

which has the same effect as

Point \_\_a = new Point();  
\_\_a.X = 0;  
\_\_a.Y = 1;   
Point a = \_\_a;

where \_\_a is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

public class Rectangle  
{  
 Point p1, p2;

public Point P1 { get { return p1; } set { p1 = value; } }  
 public Point P2 { get { return p2; } set { p2 = value; } }  
}

An instance of Rectangle can be created and initialized as follows:

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;

#### Collection initializers

A collection initializer specifies the elements of a collection.

collection-initializer:  
{ element-initializer-list }  
{ element-initializer-list , }

element-initializer-list:  
element-initializer  
element-initializer-list , element-initializer

element-initializer:  
non-assignment-expression  
{ expression-list }

expression-list:  
expression  
expression-list , expression

A collection initializer consists of a sequence of element initializers, enclosed by { and } tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by { and } tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment expression, to avoid ambiguity with member initializers. The non-assignment-expression production is defined in §7.18.

The following is an example of an object creation expression that includes a collection initializer:

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

The collection object to which a collection initializer is applied must be of a type that implements System.Collections.IEnumerable or a compile-time error occurs. For each specified element in order, the collection initializer invokes an Add method on the target object with the expression list of the element initializer as argument list, applying normal overload resolution for each invocation. Thus, the collection object must contain an applicable Add method for each element initializer.

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.

#### 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-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression new int[10, 20] produces an array instance of type int[,], and the array creation expression new int[10][,] produces an array of type int[][,]. Each expression in the expression list must be of type int, uint, long, or ulong, or implicitly convertible to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension must be 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 (§18.1), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second or third form, the rank of the specified array type or rank specifier must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

new int[,] {{0, 1}, {2, 3}, {4, 5}}

exactly corresponds to

new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}

An array creation expression of the third form is referred to as an implicitly typed array creation expression. It is similar to the second form, except that the element type of the array is not explicitly given, but determined as the best common type (§7.5.2.14) of the set of expressions in the array initializer. For a multidimensional array, 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 §12.6.

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The 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 (§6.1) to one of the following types is performed: int, uint, long, ulong. The first type in this list for which an implicit conversion exists is chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
* The computed values for the dimension lengths are validated as follows. If one or more of the values are less than zero, a System.OverflowException is thrown and no further steps are executed.
* An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown and no further steps are executed.
* All elements of the new array instance are initialized to their default values (§5.2).
* If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

int[][] a = new int[100][];

creates a single-dimensional array with 100 elements of type int[]. The initial value of each element is null. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

int[][] a = new int[100][5]; // Error

results in a compile-time error. Instantiation of the sub-arrays must instead be performed manually, as in

int[][] a = new int[100][];  
for (int i = 0; i < 100; i++) a[i] = new int[5];

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.

The following are examples of implicitly typed array creation expressions:

var a = new[] { 1, 10, 100, 1000 }; // int[]

var b = new[] { 1, 1.5, 2, 2.5 }; // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" }; // Error

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.

Implicitly typed array creation expressions can be combined with anonymous object initializers (§7.6.10.6) to create anonymously typed data structures. For example:

var contacts = new[] {  
 new {  
 Name = "Chris Smith",  
 PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }  
 },  
 new {  
 Name = "Bob Harris",  
 PhoneNumbers = new[] { "650-555-0199" }  
 }  
};

#### Delegate creation expressions

A delegate-creation-expression is used to create a new instance of a delegate-type.

delegate-creation-expression:  
new delegate-type ( expression )

The argument of a delegate creation expression must be a method group, an anonymous function or a value of either the compile time type dynamic or a delegate-type. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is an anonymous function it directly defines the parameters and method body of the delegate target. If the argument is a value it identifies a delegate instance of which to create a copy.

If the expression has the compile-time type dynamic, the delegate-creation-expression is dynamically bound (§7.2.2), and the rules below are applied at 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 (§6.6) from E to D.
* If E is an anonymous function, the delegate creation expression is processed in the same way as an anonymous function conversion (§6.5) from E to D.
* If E is a value, E must be compatible (§15.1) with D, and the result is a reference to a newly created delegate of type D that refers to the same invocation list as E. If E is not compatible with D, a compile-time error occurs.

The 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 (§6.6) from E to D.
* If E is an anonymous function, the delegate creation is evaluated as an anonymous function conversion from E to D (§6.5).
* If E is a value of a delegate-type:
* E is evaluated. If this evaluation causes an exception, no further steps are executed.
* If the value of E is null, a System.NullReferenceException is thrown and no further steps are executed.
* A new instance of the delegate type D is allocated. If there is not enough memory available to allocate the new instance, a System.OutOfMemoryException is thrown and no further steps are executed.
* The new delegate instance is initialized with the same invocation list as the delegate instance given by E.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When two delegates are combined or one is removed from another (§15.1), a new delegate results; no existing delegate has its contents changed.

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

delegate double DoubleFunc(double x);

class A  
{  
 DoubleFunc f = new DoubleFunc(Square);

static float Square(float x) {  
 return x \* x;  
 }

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.

#### 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-listopt }  
{ 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 { p1 = e1 , p2 = e2 , … pn = en }

declares an anonymous type of the form

class \_\_Anonymous1  
{  
 private readonly T1 f1 ;  
 private readonly T2 f2 ;  
 …  
 private readonly Tn fn ;

public \_\_Anonymous1(T1 a1, T2 a2,…, Tn an) {  
 f1 = a1 ;  
 f2 = a2 ;  
 …  
 fn = an ;  
 }

public T1 p1 { get { return f1 ; } }  
 public T2 p2 { get { return f2 ; } }  
 …  
 public Tn pn { get { return fn ; } }

public override bool Equals(object \_\_o) { … }  
 public override int GetHashCode() { … }  
}

where each Tx is the type of the corresponding expression ex. The expression used in a member-declarator must have a type. Thus, it is a compile-time error for an expression in a member-declarator to be null or an anonymous function. It is also a compile-time error for the expression to have an unsafe type.

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.

In the example

var p1 = new { Name = "Lawnmower", Price = 495.00 };  
var p2 = new { Name = "Shovel", Price = 26.95 };  
p1 = p2;

the assignment on the last line is permitted because p1 and p2 are of the same anonymous type.

The Equals and GetHashcode methods on anonymous types override the methods inherited from object, and are defined in terms of the Equals and GetHashcode of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name (§7.5.2), a member access (§7.5.4) or a base access (§7.6.8). This is called a projection initializer and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

identifier expr . identifier

are precisely equivalent to the following, respectively:

identifer = 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.

### 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-specifieropt  
identifier :: identifier generic-dimension-specifieropt  
unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:  
< commasopt >

commas:  
,  
commas ,

The first form of typeof-expression consists of a typeof keyword followed by a parenthesized type. The result of an expression of this form is the System.Type object for the indicated type. There is only one System.Type object for any given type. This means that for a type T, typeof(T) == typeof(T) is always true. The type cannot be dynamic.

The second form of typeof-expression consists of a typeof keyword followed by a parenthesized unbound-type-name. An unbound-type-name is very similar to a type-name (§3.8) except that an unbound-type-name contains generic-dimension-specifiers where a type-name contains type-argument-lists. When the operand of a typeof-expression is a sequence of tokens that satisfies the grammars of both unbound-type-name and type-name, namely when it contains neither a generic-dimension-specifier nor a type-argument-list, the sequence of tokens is considered to be a type-name. The meaning of an unbound-type-name is determined as follows:

* Convert the sequence of tokens to a type-name by replacing each generic-dimension-specifier with a type-argument-list having the same number of commas and the keyword object as each type-argument.
* Evaluate the resulting type-name, while ignoring all type parameter constraints.
* The unbound-type-name resolves to the unbound generic type associated with the resulting constructed type (§4.4.3).

The result of the typeof-expression is the System.Type object for the resulting unbound generic type.

The third form of typeof-expression consists of a typeof keyword followed by a parenthesized void keyword. The result of an expression of this form is the System.Type object that represents the absence of a type. The type object returned by typeof(void) is distinct from the type object returned for any type. This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of System.Type.

The typeof operator can be used on a type parameter. The result is the System.Type object for the run-time type that was bound to the type parameter. The typeof operator can also be used on a constructed type or an unbound generic type (§4.4.3). The System.Type object for an unbound generic type is not the same as the System.Type object of the instance type. 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, while the unbound generic type has no type arguments.

The example

using System;

class X<T>  
{  
 public static void PrintTypes() {  
 Type[] t = {  
 typeof(int),  
 typeof(System.Int32),  
 typeof(string),  
 typeof(double[]),  
 typeof(void),  
 typeof(T),  
 typeof(X<T>),  
 typeof(X<X<T>>),  
 typeof(X<>)  
 };  
 for (int i = 0; i < t.Length; i++) {  
 Console.WriteLine(t[i]);  
 }  
 }  
}

class Test  
{  
 static void Main() {  
 X<int>.PrintTypes();  
 }  
}

produces the following output:

System.Int32  
System.Int32  
System.String  
System.Double[]  
System.Void  
System.Int32  
X`1[System.Int32]  
X`1[X`1[System.Int32]]  
X`1[T]

Note that int and System.Int32 are the same type.

Also note that the result of typeof(X<>) does not depend on the type argument but the result of typeof(X<T>) does.

### The checked and unchecked operators

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

checked-expression:  
checked ( expression )

unchecked-expression:  
unchecked ( expression )

The checked operator evaluates the contained expression in a checked context, and the unchecked operator evaluates the contained expression in an unchecked context. A checked-expression or unchecked-expression corresponds exactly to a parenthesized-expression (§7.6.3), except that the contained expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the checked and unchecked statements (§8.11).

The following operations are affected by the overflow checking context established by the checked and unchecked operators and statements:

* The predefined ++ and -- unary operators (§7.6.9 and §7.7.5), when the operand is of an integral type.
* The predefined - unary operator (§7.7.2), when the operand is of an integral type.
* The predefined +, -, \*, and / binary operators (§7.8), when both operands are of integral types.
* Explicit numeric conversions (§6.2.1) from one integral type to another integral type, or from float or double to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

* In a checked context, if the operation is a constant expression (§7.19), a compile-time error occurs. Otherwise, when the operation is performed at 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 (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 (expressions that can be fully evaluated at compile-time), the default overflow checking context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

The body of an anonymous function is not affected by checked or unchecked contexts in which the anonymous function occurs.

In the example

class Test  
{  
 static readonly int x = 1000000;  
 static readonly int y = 1000000;

static int F() {  
 return checked(x \* y); // Throws OverflowException  
 }

static int G() {  
 return unchecked(x \* y); // Returns -727379968  
 }

static int H() {  
 return x \* y; // Depends on default  
 }  
}

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At 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.

In the example

class Test  
{  
 const int x = 1000000;  
 const int y = 1000000;

static int F() {  
 return checked(x \* y); // Compile error, overflow  
 }

static int G() {  
 return unchecked(x \* y); // Returns -727379968  
 }

static int H() {  
 return x \* y; // Compile error, overflow  
 }  
}

the overflows that occur when evaluating the constant expressions in F and H cause compile-time errors to be reported because the expressions are evaluated in a checked context. An overflow also occurs when evaluating the constant expression in G, but since the evaluation takes place in an unchecked context, the overflow is not reported.

The checked and unchecked operators 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. In the example

class Test  
{  
 static int Multiply(int x, int y) {  
 return x \* y;  
 }

static int F() {  
 return checked(Multiply(1000000, 1000000));  
 }  
}

the use of checked in F does not affect the evaluation of x \* y in Multiply, so x \* y is evaluated in the default overflow checking context.

The unchecked operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

class Test  
{  
 public const int AllBits = unchecked((int)0xFFFFFFFF);

public const int HighBit = unchecked((int)0x80000000);  
}

Both of the hexadecimal constants above are of type uint. Because the constants are outside the int range, without the unchecked operator, the casts to int would produce compile-time errors.

The checked and unchecked operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands’ data types. For example, multiplying two decimals always results in an exception on overflow even within an explicitly unchecked construct. Similarly, multiplying two floats never results in an exception on overflow even within an explicitly checked construct. In addition, other operators are never affected by the mode of checking, whether default or explicit.

### Default value expressions

A default value expression is used to obtain the default value (§5.2) of a type. Typically a default value expression is used for type parameters, since it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the null literal to a type parameter unless the type parameter is known to be a reference type.)

default-value-expression:  
default ( type )

If the type in a default-value-expression evaluates at 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 (§4.1.2).

A default-value-expression is a constant expression (§7.19) if the type is a reference type or a type parameter that is known to be a reference type (§10.1.5). In addition, a default-value-expression is a constant expression if the type is one of the following value types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, or any enumeration type.

### Anonymous method expressions

An anonymous-method-expression is one of two ways of defining an anonymous function. These are further described in §7.15.

## Unary operators

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 (§7.2.2). In this case the compile-time type of the unary-expression is dynamic, and the resolution described below will take place at run-time using the run-time type of the operand.

### Unary plus operator

For an operation of the form +x, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are:

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.

### Unary minus operator

For an operation of the form –x, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined negation operators are:

* Integer negation:

int operator –(int x);  
long operator –(long x);

The result is computed by subtracting x from zero. If the value of of x is the smallest representable value of the operand type (−231 for int or −263 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 (−231) to be written as a decimal integer literal (§2.4.4.2).

If the operand of the negation operator is of type ulong, a compile-time error occurs. An exception is the rule that permits the long value −9223372036854775808 (−263) to be written as a decimal integer literal (§2.4.4.2).

* Floating-point negation:

float operator –(float x);  
double operator –(double x);

The result is the value of x with its sign inverted. If x is NaN, the result is also NaN.

* Decimal negation:

decimal operator –(decimal x);

The result is computed by subtracting x from zero. Decimal negation is equivalent to using the unary minus operator of type System.Decimal.

### Logical negation operator

For an operation of the form !x, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

bool operator !(bool x);

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.

### Bitwise complement operator

For an operation of the form ~x, unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are:

int operator ~(int x);  
uint operator ~(uint x);  
long operator ~(long x);  
ulong operator ~(ulong x);

For each of these operators, the result of the operation is the bitwise complement of x.

Every enumeration type E implicitly provides the following bitwise complement operator:

E operator ~(E x);

The result of evaluating ~x, where x is an expression of an enumeration type E with an underlying type U, is exactly the same as evaluating (E)(~(U)x), except that the conversion to E is always performed as if in an unchecked context (§7.6.12).

### Prefix increment and decrement operators

pre-increment-expression:  
++ unary-expression

pre-decrement-expression:  
-- unary-expression

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a get and a set accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (§7.3.3) is applied to select a specific operator implementation. Predefined ++ and -- operators exist for the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any enum type. The predefined ++ operators return the value produced by adding 1 to the operand, and the predefined -- operators return the value produced by subtracting 1 from the operand. In a checked context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a System.OverflowException is thrown.

The 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 selected operator is invoked with the value of x as its argument.
* The value returned by the operator is stored in the location given by the evaluation of x.
* The value returned by the operator becomes the result of the operation.
* If x is classified as a property or indexer access:
* The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent get and set accessor invocations.
* The get accessor of x is invoked.
* The selected operator is invoked with the value returned by the get accessor as its argument.
* The set accessor of x is invoked with the value returned by the operator as its value argument.
* The value returned by the operator becomes the result of the operation.

The ++ and -- operators also support postfix notation (§7.6.9). Typically, the result of x++ or x-- is the value of x before the operation, whereas the result of ++x or --x is the value of x after the operation. In either case, x itself has the same value after the operation.

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.

### Cast expressions

A cast-expression is used to explicitly convert an expression to a given type.

cast-expression:  
( type ) unary-expression

A cast-expression of the form (T)E, where T is a type and E is a unary-expression, performs an explicit conversion (§6.2) of the value of E to type T. If no explicit conversion exists from E to T, a binding-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if E denotes a variable.

The grammar for a cast-expression leads to certain syntactic ambiguities. For 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).

To resolve cast-expression ambiguities, the following rule exists: A sequence of one or more tokens (§2.3.3) 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 (§2.4.1), a literal (§2.4.4), or any keyword (§2.4.3) except as and is.

The term “correct grammar” above means only that the sequence of tokens must conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. For example, if x and y are identifiers, then x.y is correct grammar for a type, even if x.y doesn’t actually denote a type.

From the disambiguation rule it follows that, if x and y are identifiers, (x)y, (x)(y), and (x)(-y) are cast-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).

### Await expressions

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 (§10.14). Within the nearest enclosing async function, an await-expression may 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 unsafe context

Note that an await-expression cannot occur in most places within a query-expression, because those are syntactically transformed to use non-async lambda expressions.

Inside of an async function, await cannot be used as an identifier. 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.

#### 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.

#### 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 *T*, the await-expression is classified as a value of type *T*.

#### Runtime evaluation of await expressions

At runtime, 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 runtime if *a* has the compile time type dynamic, and at compile time otherwise. Let *r* denote the resumption delegate (§10.14):
* 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.

## Arithmetic operators

The \*, /, %, +, and – operators are called the arithmetic operators.

multiplicative-expression:  
unary-expression  
multiplicative-expression \* unary-expression  
multiplicative-expression / unary-expression  
multiplicative-expression % unary-expression

additive-expression:  
multiplicative-expression  
additive-expression + multiplicative-expression  
additive-expression – multiplicative-expression

If an operand of an arithmetic operator has the compile-time type dynamic, then the expression is dynamically bound (§7.2.2). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

### Multiplication operator

For an operation of the form x \* y, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of x and y.

* Integer multiplication:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);

In a checked context, if the product is outside the range of the result type, a System.OverflowException is thrown. In an unchecked context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

* Floating-point multiplication:

float operator \*(float x, float y);  
double operator \*(double x, double y);

The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN’s. In the table, x and y are positive finite values. z is the result of x \* y. If the result is too large for the destination type, z is infinity. If the result is too small for the destination type, z is zero.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +0 | –0 | +∞ | –∞ | NaN |
| –x | –z | +z | –0 | +0 | –∞ | +∞ | NaN |
| +0 | +0 | –0 | +0 | –0 | NaN | NaN | NaN |
| –0 | –0 | +0 | –0 | +0 | NaN | NaN | NaN |
| +∞ | +∞ | –∞ | NaN | NaN | +∞ | –∞ | NaN |
| –∞ | –∞ | +∞ | NaN | NaN | –∞ | +∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Decimal multiplication:

decimal operator \*(decimal x, decimal y);

If the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. If the result value is too small to represent in the decimal format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type System.Decimal.

### Division operator

For an operation of the form x / y, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of x and y.

* Integer division:

int operator /(int x, int y);  
uint operator /(uint x, uint y);  
long operator /(long x, long y);  
ulong operator /(ulong x, ulong y);

If the value of the right operand is zero, a System.DivideByZeroException is thrown.

The division rounds the result 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 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 IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN’s. In the table, x and y are positive finite values. z is the result of x / y. If the result is too large for the destination type, z is infinity. If the result is too small for the destination type, z is zero.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +∞ | –∞ | +0 | –0 | NaN |
| –x | –z | +z | –∞ | +∞ | –0 | +0 | NaN |
| +0 | +0 | –0 | NaN | NaN | +0 | –0 | NaN |
| –0 | –0 | +0 | NaN | NaN | –0 | +0 | NaN |
| +∞ | +∞ | –∞ | +∞ | –∞ | NaN | NaN | NaN |
| –∞ | –∞ | +∞ | –∞ | +∞ | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Decimal division:

decimal operator /(decimal x, decimal y);

If the value of the right operand is zero, a System.DivideByZeroException is thrown. If the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. If the result value is too small to represent in the decimal format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representantable decimal value to the true mathematical result.

Decimal division is equivalent to using the division operator of type System.Decimal.

### Remainder operator

For an operation of the form x % y, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between x and y.

* Integer remainder:

int operator %(int x, int y);  
uint operator %(uint x, uint y);  
long operator %(long x, long y);  
ulong operator %(ulong x, ulong y);

The result of x % y is the value produced by x – (x / y) \* y. If y is zero, a System.DivideByZeroException is thrown.

If the left operand is the smallest int or long value and the right operand is -1, a System.OverflowException is thrown. In no case does x % y throw an exception where x / y would not throw an exception.

* Floating-point remainder:

float operator %(float x, float y);  
double operator %(double x, double y);

The following table lists the results of all possible combinations of 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 IEEE 754 definition (in which n is the integer closest to x / y).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | +z | NaN | NaN | x | x | NaN |
| –x | –z | –z | NaN | NaN | –x | –x | NaN |
| +0 | +0 | +0 | NaN | NaN | +0 | +0 | NaN |
| –0 | –0 | –0 | NaN | NaN | –0 | –0 | NaN |
| +∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| –∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Decimal remainder:

decimal operator %(decimal x, decimal y);

If the value of the right operand is zero, a System.DivideByZeroException is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of x.

Decimal remainder is equivalent to using the remainder operator of type System.Decimal.

### Addition operator

For an operation of the form x + y, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or both operands are of type string, the predefined addition operators concatenate the string representation of the operands.

* Integer addition:

int operator +(int x, int y);  
uint operator +(uint x, uint y);  
long operator +(long x, long y);  
ulong operator +(ulong x, ulong y);

In a checked context, if the sum is outside the range of the result type, a System.OverflowException is thrown. In an unchecked context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

* Floating-point addition:

float operator +(float x, float y);  
double operator +(double x, double y);

The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of 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.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | +∞ | –∞ | NaN |
| +0 | y | +0 | +0 | +∞ | –∞ | NaN |
| –0 | y | +0 | –0 | +∞ | –∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | +∞ | NaN | NaN |
| –∞ | –∞ | –∞ | –∞ | NaN | –∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Decimal addition:

decimal operator +(decimal x, decimal y);

If the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

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 argument is converted to its string representation by invoking the virtual ToString method inherited from type object. If ToString returns null, an empty string is substituted.

using System;

class Test  
{  
 static void Main() {  
 string s = null;  
 Console.WriteLine("s = >" + s + "<"); // displays s = ><  
 int i = 1;  
 Console.WriteLine("i = " + i); // displays i = 1  
 float f = 1.2300E+15F;  
 Console.WriteLine("f = " + f); // displays f = 1.23E+15  
 decimal d = 2.900m;  
 Console.WriteLine("d = " + d); // displays d = 2.900  
 }  
}

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a null value. A System.OutOfMemoryException may be thrown if there is not enough memory available to allocate the resulting string.

* Delegate combination. Every delegate type implicitly provides the following predefined operator, where D is the delegate type:

D operator +(D x, D y);

The binary + operator performs delegate combination when both operands are of some delegate type D. (If the operands have different delegate types, a binding-time error occurs.) If the first operand is null, the result of the operation is the value of the second operand (even if that is also null). Otherwise, if the second operand is null, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see §7.8.5 and §15.4. Since System.Delegate is not a delegate type, operator + is not defined for it.

### Subtraction operator

For an operation of the form x – y, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined subtraction operators are listed below. The operators all subtract y from x.

* Integer subtraction:

int operator –(int x, int y);  
uint operator –(uint x, uint y);  
long operator –(long x, long y);  
ulong operator –(ulong x, ulong y);

In a checked context, if the difference is outside the range of the result type, a System.OverflowException is thrown. In an unchecked context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

* Floating-point subtraction:

float operator –(float x, float y);  
double operator –(double x, double y);

The difference is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and 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.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | –∞ | +∞ | NaN |
| +0 | –y | +0 | +0 | –∞ | +∞ | NaN |
| –0 | –y | –0 | +0 | –∞ | +∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | NaN | +∞ | NaN |
| –∞ | –∞ | –∞ | –∞ | –∞ | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Decimal subtraction:

decimal operator –(decimal x, decimal y);

If the resulting value is too large to represent in the decimal format, a System.OverflowException is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type System.Decimal.

* Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where E is the enum type, and U is the underlying type of E:

U operator –(E x, E y);

This operator is evaluated exactly as (U)((U)x – (U)y). In other words, the operator computes the difference between the ordinal values of x and y, and the type of the result is the underlying type of the enumeration.

E operator –(E x, U y);

This operator is evaluated exactly as (E)((U)x – y). In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

* Delegate removal. Every delegate type implicitly provides the following predefined operator, where D is the delegate type:

D operator –(D x, D y);

The binary – operator performs delegate removal when both operands are of some delegate type D. If the operands have different delegate types, a binding-time error occurs. If the first operand is null, the result of the operation is null. Otherwise, if the second operand is null, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists (§15.1) having one or more entries, and the result is a new invocation list consisting of the first operand’s list with the second operand’s entries removed from it, provided the second operand’s list is a proper contiguous sublist of the first’s. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator (§7.10.8).) Otherwise, the result is the value of the left operand. Neither of the operands’ lists is changed in the process. If the second operand’s list matches multiple sublists of contiguous entries in the first operand’s list, the right-most matching sublist of contiguous entries is removed. If removal results in an empty list, the result is null. For example:

delegate void D(int x);

class C  
{  
 public static void M1(int i) { /\* … \*/ }  
 public static void M2(int i) { /\* … \*/ }  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1);  
 D cd2 = new D(C.M2);  
 D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1; // => M1 + M2 + M2

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1 + cd2; // => M2 + M1

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd2 + cd2; // => M1 + M1

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd2 + cd1; // => M1 + M2

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1 + cd1; // => M1 + M2 + M2 + M1  
 }  
}

## Shift operators

The << and >> operators are used to perform bit shifting operations.

shift-expression:  
additive-expression   
shift-expression << additive-expression  
shift-expression right-shift additive-expression

If an operand of a shift-expression has the compile-time type dynamic, then the expression is dynamically bound (§7.2.2). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type dynamic.

For an operation of the form x << count or x >> count, binary operator overload resolution (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be int.

The predefined shift operators are listed below.

* Shift left:

int operator <<(int x, int count);  
uint operator <<(uint x, int count);  
long operator <<(long x, int count);  
ulong operator <<(ulong x, int count);

The << operator shifts x left by a number of bits computed as described below.

The high-order bits outside the range of the result type of x are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

* Shift right:

int operator >>(int x, int count);  
uint operator >>(uint x, int count);  
long operator >>(long x, int count);  
ulong operator >>(ulong x, int count);

The >> operator shifts x right by a number of bits computed as described below.

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. For example, if x is a variable of type int, the operation unchecked((int)((uint)x >> y)) performs a logical shift right of x.

## Relational and type-testing operators

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 §7.10.10 and the as operator is described in §7.10.11.

The ==, !=, <, >, <= and >= operators are comparison operators.

If an operand of a comparison operator has the compile-time type dynamic, then the expression is dynamically bound (§7.2.2). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at 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 (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined comparison operators are described in the following sections. All predefined comparison operators return a result of type bool, as described in the following table.

|  |  |
| --- | --- |
| **Operation** | **Result** |
| x == y | true if x is equal to y, false otherwise |
| x != y | true if x is not equal to y, false otherwise |
| x < y | true if x is less than y, false otherwise |
| x > y | true if x is greater than y, false otherwise |
| x <= y | true if x is less than or equal to y, false otherwise |
| x >= y | true if x is greater than or equal to y, false otherwise |

### 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 x, ulong y);

bool operator !=(int x, int y);  
bool operator !=(uint x, uint y);  
bool operator !=(long x, long y);  
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);  
bool operator <(uint x, uint y);  
bool operator <(long x, long y);  
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);  
bool operator >(uint x, uint y);  
bool operator >(long x, long y);  
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);  
bool operator <=(uint x, uint y);  
bool operator <=(long x, long y);  
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);  
bool operator >=(uint x, uint y);  
bool operator >=(long x, long y);  
bool operator >=(ulong x, ulong y);

Each of these operators compares the numeric values of the two integer operands and returns a bool value that indicates whether the particular relation is true or false.

### 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 <=(double x, double y);

bool operator >=(float x, float y);  
bool operator >=(double x, double y);

The operators compare the operands according to the rules of the IEEE 754 standard:

* If either operand is NaN, the result is false for all operators except !=, for which the result is true. For any two operands, x != y always produces the same result as !(x == y). However, when one or both operands are NaN, the <, >, <=, and >= operators do not produce the same results as the logical negation of the opposite operator. For example, if either of x and y is NaN, then x < y is false, but !(x >= y) is true.
* When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

–∞ < –max < ... < –min < –0.0 == +0.0 < +min < ... < +max < +∞

where min and max are 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.
* A positive infinity is considered greater than all other values, but equal to another positive infinity.

### Decimal comparison operators

The predefined decimal comparison operators are:

bool operator ==(decimal x, decimal y);

bool operator !=(decimal x, decimal y);

bool operator <(decimal x, decimal y);

bool operator >(decimal x, decimal y);

bool operator <=(decimal x, decimal y);

bool operator >=(decimal x, decimal y);

Each of these operators compares the numeric values of the two decimal operands and returns a bool value that indicates whether the particular relation is true or false. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type System.Decimal.

### 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 ^ operator.

### 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.

### Reference type equality operators

The predefined reference type equality operators are:

bool operator ==(object x, object y);

bool operator !=(object x, object y);

The operators return the result of comparing the two references for equality or non-equality.

Since the predefined reference type equality operators accept operands of type object, they apply to all types that do not declare applicable operator == and operator != members. Conversely, any applicable user-defined equality operators effectively hide the predefined reference type equality operators.

The predefined reference type equality operators require one of the following:

* Both operands are a value of a type known to be a reference-type or the literal null. Furthermore, an explicit reference conversion (§6.2.4) exists from the type of either operand to the type of the other operand.
* One operand is a value of type T where T is a type-parameter and the other operand is the literal null. Furthermore T does not have the value type constraint.

Unless one of these conditions are true, a binding-time error occurs. Notable implications of these rules are:

* It is a binding-time error to use the predefined reference type equality operators to compare two references that are known to be different at binding-time. For example, if the binding-time types of the operands are two class types A and B, and if neither A nor B derives from the other, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a binding-time error.
* The predefined reference type equality operators do not permit value type operands to be compared. Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of that struct type.
* The predefined reference type equality operators never cause boxing operations to occur for their operands. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.
* If an operand of a type parameter type T is compared to null, and the run-time type of T is a value type, the result of the comparison is false.

The following example checks whether an argument of an unconstrained type parameter type is null.

class C<T>  
{  
 void F(T x) {  
 if (x == null) throw new ArgumentNullException();  
 ...  
 }  
}

The x == null construct is permitted even though T could represent a value type, and the result is simply defined to be false when T is a value type.

For an operation of the form x == y or x != y, if any applicable operator == or operator != exists, the operator overload resolution (§7.3.4) rules will select that operator instead of the predefined reference type equality operator. However, it is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type object. The example

using System;

class Test  
{  
 static void Main() {  
 string s = "Test";  
 string t = string.Copy(s);  
 Console.WriteLine(s == t);  
 Console.WriteLine((object)s == t);  
 Console.WriteLine(s == (object)t);  
 Console.WriteLine((object)s == (object)t);  
 }  
}

produces the output

True  
False  
False  
False

The s and t variables refer to two distinct string instances containing the same characters. The first comparison outputs True because the predefined string equality operator (§7.10.7) is selected when both operands are of type string. The remaining comparisons all output False because the predefined reference type equality operator is selected when one or both of the operands are of type object.

Note that the above technique is not meaningful for value types. The example

class Test  
{  
 static void Main() {  
 int i = 123;  
 int j = 123;  
 System.Console.WriteLine((object)i == (object)j);  
 }  
}

outputs False because the casts create references to two separate instances of boxed int values.

### 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. As described in §7.10.6, the reference type equality operators can be used to compare string references instead of string values.

### Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators:

bool operator ==(System.Delegate x, System.Delegate y);

bool operator !=(System.Delegate x, System.Delegate y);

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 (§15.1), those instances are equal if and only if their invocation lists are the same length, and each entry in one’s invocation list is equal (as defined below) to the corresponding entry, in order, in the other’s invocation list.

The following rules govern the equality of invocation list entries:

* If two invocation list entries both refer to the same static method then the entries are equal.
* If two invocation list entries both refer to the same non-static method on the same target object (as defined by the reference equality operators) then the entries are equal.
* Invocation list entries produced from evaluation of semantically identical anonymous-function-expressions with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.

### Equality operators and null

The == and != operators permit one operand to be a value of a nullable type and the other to be the null literal, even if no predefined or user-defined operator (in unlifted or lifted form) exists for the operation.

For an operation of one of the forms

x == null null == x x != null null != x

where x is an expression of a nullable type, if operator overload resolution (§7.2.4) fails to find an applicable operator, the result is instead computed from the HasValue property of x. Specifically, the first two forms are translated into !x.HasValue, and last two forms are translated into x.HasValue.

### The is operator

The is operator is used to dynamically check if the run-time type of an object is compatible with a given type. The result of the operation E is T, where E is an expression and T is a type, is a boolean value indicating whether E can successfully be converted to type T by a reference conversion, a boxing conversion, or an unboxing conversion. The operation is evaluated as follows, after type arguments have been substituted for all type parameters:

* If E is an anonymous function, a compile-time error occurs
* If E is a method group or the null literal, of if the type of E is a reference type or a nullable type and the value of E is null, the result is false.
* Otherwise, let D represent the dynamic type of E as follows:
* If the type of E is a reference type, D is the run-time type of the instance reference by E.
* If the type of E is a nullable type, D is the underlying type of that nullable type.
* If the type of E is a non-nullable value type, D is the type of E.
* The result of the operation depends on D and T as follows:
* If T is a reference type, the result is true if D and T are the same type, if D is a reference type and an implicit reference conversion from D to T exists, or if D is a value type and a boxing conversion from D to T exists.
* If T is a nullable type, the result is true if D is the underlying type of T.
* If T is a non-nullable value type, the result is true if D and T are the same type.
* Otherwise, the result is false.

Note that user defined conversions, are not considered by the is operator.

### The as operator

The as operator is used to explicitly convert a value to a given reference type or nullable type. Unlike a cast expression (§7.7.6), the as operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is null.

In an operation of the form E as T, E must be an expression and T must be a reference type, a type parameter known to be a reference type, or a nullable type. Furthermore, at least one of the following must be true, or otherwise a compile-time error occurs:

* An identity (§6.1.1), implicit nullable (§6.1.4), implicit reference (§6.1.6), boxing (§6.1.7), explicit nullable (§6.2.3), explicit reference (§6.2.4), or unboxing (§6.2.5) conversion exists from E to T.
* The type of E or T is an open type.
* E is the null literal.

If the compile-time type of E is not dynamic, the operation E as T produces the same result as

E is T ? (T)(E) : (T)null

except that E is only evaluated once. The compiler can be expected to optimize E as T to perform at most one dynamic type check as opposed to the two dynamic type checks implied by the expansion above.

If the compile-time type of E is dynamic, unlike the cast operator the as operator is not dynamically bound (§7.2.2). Therefore the expansion in this case is:

E is T ? (T)(object)(E) : (T)null

Note that some conversions, such as user defined conversions, are not possible with the as operator and should instead be performed using cast expressions.

In the example

class X  
{

public string F(object o) {  
 return o as string; // 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.

## Logical operators

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 (§7.2.2). 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 (§7.3.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

### 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 |(ulong x, ulong y);

int operator ^(int x, int y);  
uint operator ^(uint x, uint y);  
long operator ^(long x, long y);  
ulong operator ^(ulong x, ulong y);

The & operator computes the bitwise logical AND of the two operands, the | operator computes the bitwise logical OR of the two operands, and the ^ operator computes the bitwise logical exclusive OR of the two operands. No overflows are possible from these operations.

### 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)x op (U)y). In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

### Boolean logical operators

The predefined boolean logical operators are:

bool operator &(bool x, bool y);

bool operator |(bool x, bool y);

bool operator ^(bool x, bool y);

The result of x & y is true if both x and y are true. Otherwise, the result is false.

The result of x | y is true if either x or y is true. Otherwise, the result is false.

The result of x ^ y is true if x is true and y is false, or x is false and y is true. Otherwise, the result is false. When the operands are of type bool, the ^ operator computes the same result as the != operator.

### Nullable boolean logical operators

The nullable boolean type bool? can represent three values, true, false, and null, and is conceptually similar to the three-valued type used for boolean expressions in SQL. To ensure that the results produced by the & and | operators for bool? operands are consistent with SQL’s three-valued logic, the following predefined operators are provided:

bool? operator &(bool? x, bool? y);

bool? operator |(bool? x, bool? y);

The following table lists the results produced by these operators for all combinations of the values true, false, and null.

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | x & y | x | y |
| true | true | true | true |
| true | false | false | true |
| true | null | null | true |
| false | true | false | true |
| false | false | false | false |
| false | null | false | null |
| null | true | null | true |
| null | false | false | null |
| null | null | null | null |

## Conditional logical operators

The && and || operators are called the conditional logical operators. They are also called the “short-circuiting” logical operators.

conditional-and-expression:  
inclusive-or-expression  
conditional-and-expression && inclusive-or-expression

conditional-or-expression:  
conditional-and-expression  
conditional-or-expression || conditional-and-expression

The && and || operators are conditional versions of the & and | operators:

* The operation x && y corresponds to the operation x & y, except that y is evaluated only if x is not false.
* The operation x || y corresponds to the operation x | y, except that y is evaluated only if x is not true.

If an operand of a conditional logical operator has the compile-time type dynamic, then the expression is dynamically bound (§7.2.2). In this case the compile-time type of the expression is dynamic, and the resolution described below will take place at 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 (§7.3.4) as if the operation was written x & y or x | y. Then,

* If overload resolution fails to find a single best operator, or if overload resolution selects one of the predefined integer logical operators, a binding-time error occurs.
* Otherwise, if the selected operator is one of the predefined boolean logical operators (§7.11.3) or nullable boolean logical operators (§7.11.4), the operation is processed as described in §7.12.1.
* Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in §7.12.2.

It is not possible to directly overload the conditional logical operators. However, because the conditional logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical operators are, with certain restrictions, also considered overloads of the conditional logical operators. This is described further in §7.12.2.

### 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.

### User-defined conditional logical operators

When the operands of && or || are of types that declare an applicable user-defined operator & or operator |, both of the following must be true, where T is the type in which the selected operator is declared:

* The return type and the type of each parameter of the selected operator must be T. In other words, the operator must compute the logical AND or the logical OR of two operands of type T, and must return a result of type T.
* T must contain declarations of operator true and operator false.

A binding-time error occurs if either of these requirements is not satisfied. Otherwise, the && or || operation is evaluated by combining the user-defined operator true or operator false with the selected user-defined operator:

* The operation x && y is evaluated as T.false(x) ? x : T.&(x, y), where T.false(x) is an invocation of the operator false declared in T, and T.&(x, y) is an invocation of the selected operator &. In other words, x is first evaluated and operator false is invoked on the result to determine if x is definitely false. Then, if x is definitely false, the result of the operation is the value previously computed for x. Otherwise, y is evaluated, and the selected operator & is invoked on the value previously computed for x and the value computed for y to produce the result of the operation.
* The operation x || y is evaluated as T.true(x) ? x : T.|(x, y), where T.true(x) is an invocation of the operator true declared in T, and T.|(x, y) is an invocation of the selected operator |. In other words, x is first evaluated and operator true is invoked on the result to determine if x is definitely true. Then, if x is definitely true, the result of the operation is the value previously computed for x. Otherwise, y is evaluated, and the selected operator | is invoked on the value previously computed for x and the value computed for y to produce the result of the operation.

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.

For an example of a type that implements operator true and operator false, see §11.4.2.

## The null coalescing operator

The ?? operator is called the null coalescing operator.

null-coalescing-expression:  
conditional-or-expression  
conditional-or-expression ?? null-coalescing-expression

A null coalescing expression of the form a ?? b requires a to be of a nullable type or reference type. If a is non-null, the result of a ?? b is a; otherwise, the result is b. The operation evaluates b only if a is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form a ?? b ?? c is evaluated as a ?? (b ?? c). In general terms, an expression of the form E1 ?? E2 ?? ... ?? EN returns the first of the operands that is non-null, or null if all operands are null.

The type of the expression a ?? b depends on which implicit conversions are available on the operands. In order of preference, the type of a ?? b is A0, 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 A0 is the underlying type of A if A is a nullable type, or A otherwise. Specifically, a ?? b is processed as follows:

* If A exists and is not a nullable type or a reference type, a compile-time error occurs.
* If b is a dynamic expression, the result type is dynamic. At 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 type and an implicit conversion exists from b to A0, the result type is A0. At run-time, a is first evaluated. If a is not null, a is unwrapped to type A0, and this becomes the result. Otherwise, b is evaluated and converted to type A0, 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 b has a type B and an implicit conversion exists from a to B, the result type is B. At run-time, a is first evaluated. If a is not null, a is unwrapped to type A0 (if A exists and is nullable) and 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.

## 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. For example, an expression of the form a ? b : c ? d : e is evaluated as a ? b : (c ? d : e).

The first operand of the ?: operator must be an expression that can be implicitly converted to bool, or an expression of a type that implements operator true. If neither of these requirements is satisfied, a compile-time error occurs.

The second and third operands, x and y, of the ?: operator control the type of the conditional expression.

* If x has type X and y has type Y then
* If an implicit conversion (§6.1) exists from X to Y, but not from Y to X, then Y is the type of the conditional expression.
* If an implicit conversion (§6.1) exists from Y to X, but not from X to Y, then X is the type of the conditional expression.
* Otherwise, no expression type can be determined, and a compile-time error occurs.
* If only one of x and y has a type, and both x and y, of areimplicitly 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.

## Anonymous function expressions

An anonymous function is an expression that represents an “in-line” method definition. An anonymous function does not have a value or type in and of itself, but is convertible to a compatible delegate or expression tree type. The evaluation of an anonymous function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method which the anonymous function defines. If it is an expression tree type, the conversion evaluates to an expression tree which represents the structure of the method as an object structure.

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.

lambda-expression:  
asyncopt anonymous-function-signature => anonymous-function-body

anonymous-method-expression:  
asyncopt delegate explicit-anonymous-function-signatureopt 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-modifieropt type identifier

anonymous-function-parameter-modifier:   
ref  
out

implicit-anonymous-function-signature:  
( implicit-anonymous-function-parameter-listopt )  
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 §10.14.

The parameters of an anonymous function in the form of a lambda-expression can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types (§6.5).

In an anonymous function with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

( param ) => expr

can be abbreviated to

param => expr

The parameter list of an anonymous function in the form of an anonymous-method-expression is optional. If given, the parameters must be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing out parameters.

A block body of an anonymous function is reachable (§8.1) unless the anonymous function occurs inside an unreachable statement.

Some examples of anonymous functions follow below:

x => x + 1 // Implicitly typed, expression body

x => { return x + 1; } // Implicitly typed, statement body

(int x) => x + 1 // Explicitly typed, expression body

(int x) => { return x + 1; } // Explicitly typed, statement body

(x, y) => x \* y // Multiple parameters

() => Console.WriteLine() // No parameters

async (t1,t2) => await t1 + await t2 // Async

delegate (int x) { return x + 1; } // Anonymous method expression

delegate { return 1 + 1; } // Parameter list omitted

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 must be a statement block.
* Only lambda-expressions have conversions to compatible expression tree types (§4.6).

### Anonymous function signatures

The optional anonymous-function-signature of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the anonymous-function-body. (§3.7) Together with the parameter list (if given) the anonymous-method-body constitutes a declaration space (§3.3). It is thus a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant or parameter whose scope includes the anonymous-method-expression or lambda-expression.

If an anonymous function has an explicit-anonymous-function-signature, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order. In contrast to method group conversions (§6.6), 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 (§4.6).

### Anonymous function bodies

The body (expression or block) of an anonymous function is subject to the following rules:

* If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature it can be converted to a delegate type or expression type having parameters (§6.5), but the parameters cannot be accessed in the body.
* Except for ref or out parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a ref or out parameter.
* When the type of this is a struct type, it is a compile-time error for the body to access this. This is true whether the access is explicit (as in this.x) or implicit (as in x where x is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
* The body has access to the outer variables (§7.15.5) of the anonymous function. Access 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 (§7.15.6).
* It is a compile-time error for the body to contain a goto statement, break statement, or continue statement whose target is outside the body or within the body of a contained anonymous function.
* A return statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member. An expression specified in a return statement must be implicitly convertible to the return type of the delegate type or expression tree type to which the nearest enclosing lambda-expression or anonymous-method-expression is converted (§6.5).

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the lambda-expression or anonymous-method-expression. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements must be of a form reserved for compiler use.

### Overload resolution

Anonymous functions in an argument list participate in type inference and overload resolution. Please refer to §7.5.2 and §7.5.3 for the exact rules.

The following example illustrates the effect of anonymous functions on overload resolution.

class ItemList<T>: List<T>  
{  
 public int Sum(Func<T,int> selector) {  
 int sum = 0;  
 foreach (T item in this) sum += selector(item);  
 return sum;  
 }

public double Sum(Func<T,double> selector) {  
 double sum = 0;  
 foreach (T item in this) sum += selector(item);  
 return sum;  
 }  
}

The ItemList<T> class has two Sum methods. Each takes a selector argument, which extracts the value to sum over from a list item. The extracted value can be either an int or a double and the resulting sum is likewise either an int or a double.

The Sum methods could for example be used to compute sums from a list of detail lines in an order.

class Detail  
{  
 public int UnitCount;  
 public double UnitPrice;  
 ...  
}

void ComputeSums() {  
 ItemList<Detail> orderDetails = GetOrderDetails(...);  
 int totalUnits = orderDetails.Sum(d => d.UnitCount);  
 double orderTotal = orderDetails.Sum(d => d.UnitPrice \* d.UnitCount);  
 ...  
}

In the first invocation of orderDetails.Sum, both Sum methods are applicable because the anonymous function d => d.UnitCount is compatible with both Func<Detail,int> and Func<Detail,double>. However, overload resolution picks the first Sum method because the conversion to Func<Detail,int> is better than the conversion to Func<Detail,double>.

In the second invocation of orderDetails.Sum, only the second Sum method is applicable because the anonymous function d => d.UnitPrice \* d.UnitCount produces a value of type double. Thus, overload resolution picks the second Sum method for that invocation.

### Anonymous functions and dynamic binding

An anonymous function cannot be a receiver, argument or operand of a dynamically bound operation.

### Outer variables

Any local variable, value parameter, or parameter array whose scope includes the lambda-expression or anonymous-method-expression is called an outer variable of the anonymous function. In an instance function member of a class, the this value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

#### Captured outer variables

When an outer variable is referenced by an anonymous function, the outer variable is said to have been captured by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated (§5.1.7). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

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 (which doesn’t happen until the very end of the program). Since each invocation of the anonymous function operates on the same instance of x, the output of the example is:

1  
2  
3

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable (§18.3), but is instead considered to be a moveable variable. Thus any unsafe code that takes the address of a captured outer variable must first use the fixed statement to fix the variable.

Note that unlike an uncaptured variable, a captured local variable can be simultaneously exposed to multiple threads of execution.

#### Instantiation of local variables

A local variable is considered to be instantiated when execution enters the scope of the variable. For example, when the following method is invoked, the local variable x is instantiated and initialized three times—once for each iteration of the loop.

static void F() {  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 ...  
 }  
}

However, moving the declaration of x outside the loop results in a single instantiation of x:

static void F() {  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 ...  
 }  
}

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 instantation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

The example

using System;

delegate void D();

class Test  
{  
 static D[] F() {  
 D[] result = new D[3];  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 result[i] = () => { Console.WriteLine(x); };  
 }  
 return result;  
 }

static void Main() {  
 foreach (D d in F()) d();  
 }  
}

produces the output:

1  
3  
5

However, when the declaration of x is moved outside the loop:

static D[] F() {  
 D[] result = new D[3];  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 result[i] = () => { Console.WriteLine(x); };  
 }  
 return result;  
}

the output is:

5  
5  
5

If a for-loop declares an iteration variable, that variable itself is considered to be declared outside of the loop. Thus, if the example is changed to capture the iteration variable itself:

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

It is possible for anonymous function delegates to share some captured variables yet have separate instances of others. For example, if F is changed to

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:

1 1  
2 1  
3 1

Separate anonymous functions can capture the same instance of an outer variable. 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

### Evaluation of anonymous function expressions

An anonymous function F must always be converted to a delegate type D or an expression tree type E, either directly or through the execution of a delegate creation expression new D(F). This conversion determines the result of the anonymous function, as described in §6.5.

## Query expressions

Query expressions provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

query-expression:  
from-clause query-body

from-clause:  
from typeopt identifier in expression

query-body:  
query-body-clausesopt select-or-group-clause query-continuationopt

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 typeopt 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-directionopt

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 can be followed by zero or more from, let, where, join or orderby clauses. Each from clause is a generator introducing a range variable which 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.

### Ambiguities in query expressions

Query expressions contain a number of “contextual keywords”, i.e., identifiers that have special meaning in a given context. Specifically these are from, where, join, on, equals, into, let, orderby, ascending, descending, select, group and by. In order to avoid ambiguities in query expressions caused by mixed use of these identifiers as keywords or simple names, these identifiers are considered keywords when occurring anywhere within a query expression.

For this purpose, a query expression is any expression that starts with “from identifier” *followed by any token except “*;*”, “*=*” or “*,*”.*

*In order to use these words as identifiers within a query expression, they can be prefixed with “*@*” (§2.4.2).*

### Query expression translation

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the *query expression pattern* (§7.16.3). Specifically, query expressions are translated into invocations of methods named Where, Select, SelectMany, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, GroupBy, and Cast.These methods are expected to have particular signatures and result types, as described in §7.16.3. These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

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.

Assignment to range variables is not allowed in query expressions. However a C# implementation is permitted to not always enforce this restriction, since this may sometimes not be possible with the syntactic translation scheme presented here.

Certain translations inject range variables with transparent identifiers denoted by \*. The special properties of transparent identifiers are discussed further in §7.16.2.7.

#### Select and groupby clauses with continuations

A query expression with a continuation

from … into x …

is translated into

from x in ( from … ) …

The translations in the following sections assume that queries have no into continuations.

The example

from c in customers  
group c by c.Country into g  
select new { Country = g.Key, CustCount = g.Count() }

is translated into

from g in  
 from c in customers  
 group c by c.Country  
select new { Country = g.Key, CustCount = g.Count() }

the final translation of which is

customers.  
GroupBy(c => c.Country).  
Select(g => new { Country = g.Key, CustCount = g.Count() })

#### Explicit range variable types

A from clause that explicitly specifies a range variable type

from T x in e

is translated into

from x in ( e ) . Cast < T > ( )

A join clause that explicitly specifies a range variable type

join T x in e on k1 equals k2

is translated into

join x in ( e ) . Cast < T > ( ) on k1 equals k2

The translations in the following sections assume that queries have no explicit range variable types.

The example

from Customer c in customers  
where c.City == "London"  
select c

is translated into

from c in customers.Cast<Customer>()  
where c.City == "London"  
select c

the final translation of which is

customers.  
Cast<Customer>().  
Where(c => c.City == "London")

Explicit range variable types are useful for querying collections that implement the non-generic IEnumerable interface, but not the generic IEnumerable<T> interface. In the example above, this would be the case if customers were of type ArrayList.

#### Degenerate query expressions

A query expression of the form

from x in e select x

is translated into

( e ) . Select ( x => x )

The example

from c in customers  
select c

Is translated into

customers.Select(c => c)

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. It is important however to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling Select on the source. It is then up to the implementers of Select and other query operators to ensure that these methods never return the source object itself.

#### From, let, where, join and orderby clauses

A query expression with a second from clause followed by a select clause

from x1 in e1  
from x2 in e2  
select v

is translated into

( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )

A query expression with a second from clause followed by something other than a select clause:

from x1 in e1  
from x2 in e2  
…

is translated into

from \* in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )  
…

A query expression with a let clause

from x in e  
let y = f  
…

is translated into

from \* in ( e ) . Select ( x => new { x , y = f } )  
…

A query expression with a where clause

from x in e  
where f  
…

is translated into

from x in ( e ) . Where ( x => f )  
…

A query expression with a join clause without an into followed by a select clause

from x1 in e1  
join x2 in e2 on k1 equals k2  
select v

is translated into

( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )

A query expression with a join clause without an into followed by something other than a select clause

from x1 in e1  
join x2 in e2 on k1 equals k2   
…

is translated into

from \* in ( e1 ) . Join(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })  
…

A query expression with a join clause with an into followed by a select clause

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
select v

is translated into

( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )

A query expression with a join clause with an into followed by something other than a select clause

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
…

is translated into

from \* in ( e1 ) . GroupJoin(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })  
…

A query expression with an orderby clause

from x in e  
orderby k1 , k2 , … , kn  
…

is translated into

from x in ( e ) .   
OrderBy ( x => k1 ) .   
ThenBy ( x => k2 ) .  
 … .   
ThenBy ( x => kn )  
…

If an ordering clause specifies a descending direction indicator, an invocation of OrderByDescending or ThenByDescending is produced instead.

The following translations assume that there are no let, where, join or orderby clauses, and no more than the one initial from clause in each query expression.

The example

from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }

is translated into

customers.  
SelectMany(c => c.Orders,  
 (c,o) => new { c.Name, o.OrderID, o.Total }  
)

The example

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

is translated into

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

the final translation of which is

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })

where x is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

from o in orders  
let t = o.Details.Sum(d => d.UnitPrice \* d.Quantity)  
where t >= 1000  
select new { o.OrderID, Total = t }

is translated into

from \* in orders.  
 Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) })  
where t >= 1000   
select new { o.OrderID, Total = t }

the final translation of which is

orders.  
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) }).  
Where(x => x.t >= 1000).  
Select(x => new { x.o.OrderID, Total = x.t })

where x is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
select new { c.Name, o.OrderDate, o.Total }

is translated into

customers.Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c.Name, o.OrderDate, o.Total })

The example

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID into co  
let n = co.Count()  
where n >= 10  
select new { c.Name, OrderCount = n }

is translated into

from \* in customers.  
 GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c, co })  
let n = co.Count()  
where n >= 10   
select new { c.Name, OrderCount = n }

the final translation of which is

customers.  
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c, co }).  
Select(x => new { x, n = x.co.Count() }).  
Where(y => y.n >= 10).  
Select(y => new { y.x.c.Name, OrderCount = y.n)

where x and y are compiler generated identifiers that are otherwise invisible and inaccessible.

The example

from o in orders  
orderby o.Customer.Name, o.Total descending  
select o

has the final translation

orders.  
OrderBy(o => o.Customer.Name).  
ThenByDescending(o => o.Total)

#### Select clauses

A query expression of the form

from x in e select v

is translated into

( e ) . Select ( x => v )

except when v is the identifier x, the translation is simply

( e )

For example

from c in customers.Where(c => c.City == “London”)  
select c

is simply translated into

customers.Where(c => c.City == “London”)

#### Groupby clauses

A query expression of the form

from x in e group v by k

is translated into

( e ) . GroupBy ( x => k , x => v )

except when v is the identifier x, the translation is

( e ) . GroupBy ( x => k )

The example

from c in customers  
group c.Name by c.Country

is translated into

customers.  
GroupBy(c => c.Country, c => c.Name)

#### Transparent identifiers

Certain translations inject range variables with transparent identifiers denoted by \*. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

* When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.
* When a member with a transparent identifier is in scope, the members of that member are in scope as well.
* When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.

In the translation steps described above, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and show how transparent identifiers can be translated away.

The example

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.Total }

is translated into

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.Total }

which is further translated into

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(\* => o.Total).  
Select(\* => new { c.Name, o.Total })

which, when transparent identifiers are erased, is equivalent to

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.Total })

where x is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
join d in details on o.OrderID equals d.OrderID  
join p in products on d.ProductID equals p.ProductID  
select new { c.Name, o.OrderDate, p.ProductName }

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.

### The query expression pattern

The Query expression pattern establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the methods of the pattern can be implemented as instance methods or as extension methods because the two have the same invocation syntax, and the methods can request delegates or expression trees because anonymous functions are convertible to both.

The recommended shape of a generic type C<T> that supports the query expression pattern is shown below. A generic type is used in order to illustrate the proper relationships between parameter and result 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 O<T> ThenByDescending<K>(Func<T,K> keySelector);  
}

class G<K,T> : C<T>  
{  
 public K Key { get; }  
}

The methods above use the generic delegate types Func<T1, R> and Func<T1, T2, R>, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between C<T> and O<T> which ensures that the ThenBy and ThenByDescending methods are available only on the result of an OrderBy or OrderByDescending. Also notice the recommended shape of the result of GroupBy—a sequence of sequences, where each inner sequence has an additional Key property.

The System.Linq namespace provides an implementation of the query operator pattern for any type that implements the System.Collections.Generic.IEnumerable<T> interface.

## Assignment operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

assignment:  
unary-expression assignment-operator expression

assignment-operator:  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The = operator is called the simple assignment operator. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in §10.8.1). The simple assignment operator is described in §7.17.1.

The assignment operators other than the = operator are called the compound assignment operators. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in §7.17.2.

The += and -= operators with an event access expression as the left operand are called the event assignment operators. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in §7.17.3.

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form a = b = c is evaluated as a = (b = c).

### 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 (§7.2.2). In this case the compile-time type of the assignment expression is dynamic, and the resolution described below will take place at run-time based on the run-time type of E.

In a simple assignment, the right operand must be an expression that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a set accessor. If this is not the case, a binding-time error occurs.

The 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 (§6.1).
* 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 (§6.1.6) exists from the actual type of the instance referenced by y to the actual element type of the array instance containing x. Otherwise, a System.ArrayTypeMismatchException is thrown.
* The value resulting from the evaluation and conversion of y is stored into the location given by the evaluation of x.
* If x is classified as a property or indexer access:
* The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent set accessor invocation.
* y is evaluated and, if required, converted to the type of x through an implicit conversion (§6.1).
* The set accessor of x is invoked with the value computed for y as its value argument.

The array co-variance rules (§12.5) permit a value of an array type A[] to be a reference to an instance of an array type B[], provided an implicit reference conversion exists from B to A. Because of these rules, assignment to an array element of a reference-type requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

string[] sa = new string[10];  
object[] oa = sa;

oa[0] = null; // Ok  
oa[1] = "Hello"; // Ok  
oa[2] = new ArrayList(); // ArrayTypeMismatchException

the last assignment causes a System.ArrayTypeMismatchException to be thrown because an instance of ArrayList cannot be stored in an element of a string[].

When a property or indexer declared in a struct-type is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a binding-time error occurs. Because of §7.6.4, the same rule also applies to fields.

Given the declarations:

struct Point  
{  
 int x, y;

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

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.

### 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 (§7.2.2). In this case the compile-time type of the assignment expression is dynamic, and the resolution described below will take place at run-time based on the run-time type of E.

An operation of the form x op= y is processed by applying binary operator overload resolution (§7.3.4) as if the operation was written x op y. Then,

* If the return type of the selected operator is implicitly convertible to the type of x, the operation is evaluated as x = x op y, except that x is evaluated only once.
* Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is explicitly convertible to the type of x, and if y is implicitly convertible to the type of x or the operator is a shift operator, then the operation is evaluated as x = (T)(x op y), where T is the type of x, except that x is evaluated only once.
* Otherwise, the compound assignment is invalid, and a binding-time error occurs.

The term “evaluated only once” means that in the evaluation of x op y, the results of any constituent expressions of x are temporarily saved and then reused when performing the assignment to x. For example, in the assignment A()[B()] += C(), where A is a method returning int[], and B and C are methods returning int, the methods are invoked only once, in the order A, B, C.

When the left operand of a compound assignment is a property access or indexer access, the property or indexer must have both a get accessor and a set accessor. If this is not the case, a binding-time error occurs.

The second rule above permits x op= y to be evaluated as x = (T)(x op y) in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type sbyte, byte, short, ushort, or char. Even when both arguments are of one of those types, the predefined operators produce a result of type int, as described in §7.3.6.2. Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that x op= y is permitted if both of x op y and x = y are permitted. In the example

byte b = 0;  
char ch = '\0';  
int i = 0;

b += 1; // Ok  
b += 1000; // Error, b = 1000 not permitted  
b += i; // Error, b = i not permitted  
b += (byte)i; // Ok

ch += 1; // Error, ch = 1 not permitted  
ch += (char)1; // Ok

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

This also means that compound assignment operations support lifted operations. In the example

int? i = 0;  
i += 1; // Ok

the lifted operator +(int?,int?) is used.

### 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 (§6.1).
* An event accessor of the event is invoked, with argument list consisting of the right operand, after evaluation and, if necessary, conversion. If the operator was +=, the add accessor is invoked; if the operator was -=, the remove accessor is invoked.

An event assignment expression does not yield a value. Thus, an event assignment expression is valid only in the context of a statement-expression (§8.6).

## 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

## Constant expressions

A constant-expression is an expression that can be fully evaluated at compile-time.

constant-expression:  
expression

A constant expression must be the null literal or a value with one of the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, object, string, or any enumeration type. Only the following constructs are permitted in constant expressions:

* Literals (including the null literal).
* References to const members of class and struct types.
* References to members of enumeration types.
* References to const parameters or local variables
* Parenthesized sub-expressions, which are themselves constant expressions.
* Cast expressions, provided the target type is one of the types listed above.
* checked and unchecked expressions
* Default value expressions
* The predefined +, –, !, and ~ unary operators.
* The predefined +, –, \*, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <=, and >= binary operators, provided each operand is of a type listed above.
* The ?: conditional operator.

The following conversions are permitted in constant expressions:

* Identity conversions
* Numeric conversions
* Enumeration conversions
* Constant expression conversions
* Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

class C {  
 const object i = 5; // error: boxing conversion not permitted  
 const object str = “hello”; // error: implicit reference conversion  
}

the initialization of iis an error because a boxing conversion is required. The initialization of str is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression 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 non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an unchecked context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (§7.19).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

* Constant declarations (§10.4).
* Enumeration member declarations (§14.3).
* Default arguments of formal parameter lists (§10.6.1)
* case labels of a switch statement (§8.7.2).
* goto case statements (§8.9.3).
* Dimension lengths in an array creation expression (§7.6.10.4) that includes an initializer.
* Attributes (§17).

An implicit constant expression conversion (§6.1.9) 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.

## Boolean expressions

A boolean-expression is an expression that yields a result of type bool; either directly or through application of operator true in certain contexts as specified in the following.

boolean-expression:  
expression

The controlling conditional expression of an if-statement (§8.7.1), while-statement (§8.8.1), do-statement (§8.8.2), or for-statement (§8.8.3) is a boolean-expression. The controlling conditional expression of the ?: operator (§7.14) follows the same rules as a boolean-expression, but for reasons of operator precedence is classified as a conditional-or-expression.

A boolean-expression E is required to be able to produce a value of type bool, as follows:

* If E is implicitly convertible to bool then at runtime that implicit conversion is applied.
* Otherwise, unary operator overload resolution (§7.3.3) is used to find a unique best implementation of operator true on E, and that implementation is applied at runtime.
* If no such operator is found, a binding-time error occurs.

The DBBool struct type in §11.4.2 provides an example of a type that implements operator true and operator false.

22

# ATTRIBUTES

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the method-modifiers public, protected, internal, and private.

C# enables programmers to invent new kinds of declarative information, called attributes. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. For instance, a framework might define a HelpAttribute attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

Attributes are defined through the declaration of attribute classes (§17.1), which may have positional and named parameters (§17.1.2). Attributes are attached to entities in a C# program using attribute specifications (§17.2), and can be retrieved at run-time as attribute instances (§17.3).

## Attribute classes

A class that derives from the abstract class System.Attribute, whether directly or indirectly, is an attribute class. The declaration of an attribute class defines a new kind of attribute that can be placed on a declaration. By convention, attribute classes are named with a suffix of Attribute. Uses of an attribute may either include or omit this suffix.

### Attribute usage

The attribute AttributeUsage (§17.4.1) is used to describe how an attribute class can be used.

AttributeUsage has a positional parameter (§17.1.2) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]  
public class SimpleAttribute: Attribute   
{  
 ...  
}

defines an attribute class named SimpleAttribute that can be placed on class-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 {...}

AttributeUsage has a named parameter (§17.1.2) called AllowMultiple, which indicates whether the attribute can be specified more than once for a given entity. If AllowMultiple for an attribute class is true, then that attribute class is a multi-use attribute class, and can be specified more than once on an entity. If AllowMultiple for an attribute class is false or it is unspecified, then that attribute class is a single-use attribute class, and can be specified at most once on an entity.

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.

AttributeUsage has another named parameter called Inherited, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If Inherited for an attribute class is true, then that attribute is inherited. If Inherited for an attribute class is false then that attribute is not inherited. If it is unspecified, its default value is true.

An attribute class X not having an AttributeUsage attribute attached to it, as in

using System;

class X: Attribute {...}

is equivalent to the following:

using System;

[AttributeUsage(  
 AttributeTargets.All,  
 AllowMultiple = false,  
 Inherited = true)  
]  
class X: Attribute {...}

### 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.

The example

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class HelpAttribute: Attribute  
{  
 public HelpAttribute(string url) { // Positional parameter  
 ...  
 }

public string Topic { // Named parameter  
 get {...}  
 set {...}  
 }

public string Url {  
 get {...}  
 }  
}

defines an attribute class named HelpAttribute that has one positional parameter, url, and one named parameter, Topic. Although it is 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  
{  
 ...  
}

### 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.
* An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility (§17.2).
* Single-dimensional arrays of the above types.

A constructor argument or public field which does not have one of these types, cannot be used as a positional or named parameter in an attribute specification.

## Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for type-declarations (§9.6), class-member-declarations (§10.1.5), interface-member-declarations (§13.2), struct-member-declarations (§11.2), enum-member-declarations (§14.3), accessor-declarations (§10.7.2), event-accessor-declarations (§10.8.1), and formal-parameter-lists (§10.6.1).

Attributes are specified in attribute sections. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications [A][B], [B][A], [A, B], and [B, A] are equivalent.

global-attributes:  
global-attribute-sections

global-attribute-sections:  
global-attribute-section  
global-attribute-sections global-attribute-section

global-attribute-section:  
[ global-attribute-target-specifier attribute-list ]  
[ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:  
global-attribute-target :

global-attribute-target:  
assembly  
module

attributes:  
attribute-sections

attribute-sections:  
attribute-section  
attribute-sections attribute-section

attribute-section:  
[ attribute-target-specifieropt attribute-list ]  
[ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier:  
attribute-target :

attribute-target:  
field  
event  
method  
param  
property  
return  
type

attribute-list:  
attribute  
attribute-list , attribute

attribute:  
attribute-name attribute-argumentsopt

attribute-name:  
 type-name

attribute-arguments:  
( positional-argument-listopt )  
( 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

An attribute consists of an attribute-name and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an attribute-argument-expression; a named argument consists of a name, followed by an equal sign, followed by an attribute-argument-expression, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

The attribute-name identifies an attribute class. If the form of attribute-name is type-name then this name must refer to an attribute class. Otherwise, a compile-time error occurs. The example

class Class1 {}

[Class1] class Class2 {} // Error

results in a compile-time error because it attempts to use Class1 as an attribute class when Class1 is not an attribute class.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an attribute-target-specifier. When an attribute is placed at the global level, a global-attribute-target-specifier is required. In all other locations, a reasonable default is applied, but an attribute-target-specifier can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, attribute-target-specifiers can be omitted except at the global level. The potentially ambiguous contexts are resolved as follows:

* An attribute specified at global scope can apply either to the target assembly or the target module. No default exists for this context, so an attribute-target-specifier is always required in this context. The presence of the assembly attribute-target-specifier indicates that the attribute applies to the target assembly; the presence of the module attribute-target-specifier indicates that the attribute applies to the target module.
* An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an attribute-target-specifier, the attribute applies to the delegate. The presence of the type attribute-target-specifier indicates that the attribute applies to the delegate; the presence of the return attribute-target-specifier indicates that the attribute applies to the return value.
* An attribute specified on a method declaration can apply either to the method being declared or to its return value. In the absence of an attribute-target-specifier, the attribute applies to the method. The presence of the method attribute-target-specifier indicates that the attribute applies to the method; the presence of the return attribute-target-specifier indicates that the attribute applies to the return value.
* An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an attribute-target-specifier, the attribute applies to the operator. The presence of the method attribute-target-specifier indicates that the attribute applies to the operator; the presence of the return attribute-target-specifier indicates that the attribute applies to the return value.
* An attribute specified on an event declaration that omits event accessors can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove methods. In the absence of an attribute-target-specifier, the attribute applies to the event. The presence of the event attribute-target-specifier indicates that the attribute applies to the event; the presence of the field attribute-target-specifier indicates that the attribute applies to the field; and the presence of the method attribute-target-specifier indicates that the attribute applies to the methods.
* An attribute specified on a get accessor declaration for a property or indexer declaration can apply either to the associated method or to its return value. In the absence of an attribute-target-specifier, the attribute applies to the method. The presence of the method attribute-target-specifier indicates that the attribute applies to the method; the presence of the return attribute-target-specifier indicates that the attribute applies to the return value.
* An attribute specified on a set accessor for a property or indexer declaration can apply either to the associated method or to its lone implicit parameter. In the absence of an attribute-target-specifier, the attribute applies to the method. The presence of the method attribute-target-specifier indicates that the attribute applies to the method; the presence of the param attribute-target-specifier indicates that the attribute applies to the parameter; the presence of the return attribute-target-specifier indicates that the attribute applies to the return value.
* An attribute specified on an add or remove accessor declaration for an event declaration can apply either to the associated method or to its lone parameter. In the absence of an attribute-target-specifier, the attribute applies to the method. The presence of the method attribute-target-specifier indicates that the attribute applies to the method; the presence of the param attribute-target-specifier indicates that the attribute applies to the parameter; the presence of the return attribute-target-specifier indicates that the attribute applies to the return value.

In other contexts, inclusion of an attribute-target-specifier is permitted but unnecessary. For instance, a class declaration may either include or omit the specifier type:

[type: Author("Brian Kernighan")]  
class Class1 {}

[Author("Dennis Ritchie")]  
class Class2 {}

It is an error to specify an invalid attribute-target-specifier. For instance, the specifier param cannot be used on a class declaration:

[param: Author("Brian Kernighan")] // Error  
class Class1 {}

By convention, attribute classes are named with a suffix of Attribute. An attribute-name of the form type-name may either include or omit this suffix. If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the attribute-name is spelled such that its right-most identifier is a verbatim identifier (§2.4.2), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

using System;

[AttributeUsage(AttributeTargets.All)]  
public class X: Attribute  
{}

[AttributeUsage(AttributeTargets.All)]  
public class XAttribute: Attribute  
{}

[X] // Error: ambiguity  
class Class1 {}

[XAttribute] // Refers to XAttribute  
class Class2 {}

[@X] // Refers to X  
class Class3 {}

[@XAttribute] // Refers to XAttribute  
class Class4 {}

shows two attribute classes named X and XAttribute. The attribute [X] is ambiguous, since it could refer to either X or XAttribute. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute [XAttribute] is not ambiguous (although it would be if there was an attribute class named XAttributeAttribute!). If the declaration for class X is removed, then both attributes refer to the attribute class named XAttribute, as follows:

using System;

[AttributeUsage(AttributeTargets.All)]  
public class XAttribute: Attribute  
{}

[X] // Refers to XAttribute  
class Class1 {}

[XAttribute] // Refers to XAttribute  
class Class2 {}

[@X] // Error: no attribute named "X"  
class Class3 {}

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class HelpStringAttribute: Attribute  
{  
 string value;

public HelpStringAttribute(string value) {  
 this.value = value;  
 }

public string Value {  
 get {...}  
 }  
}

[HelpString("Description of Class1")]  
[HelpString("Another description of Class1")]  
public class Class1 {}

results in a compile-time error because it attempts to use HelpString, which is a single-use attribute class, more than once on the declaration of Class1.

An expression E is an attribute-argument-expression if all of the following statements are true:

* The type of E is an attribute parameter type (§17.1.3).
* At compile-time, the value of E can be resolved to one of the following:
* A constant value.
* A System.Type object.
* A one-dimensional array of attribute-argument-expressions.

For 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 {}

A typeof-expression (§7.6.11) used as an attribute argument expression can reference a non-generic type, a closed constructed type, or an unbound generic type, but it cannot reference an open type. This is to ensure that the expression can be resolved at compile-time.

class A: Attribute  
{  
 public A(Type t) {...}  
}

class G<T>  
{  
 [A(typeof(T))] T t; // Error, open type in attribute  
}

class X  
{  
 [A(typeof(List<int>))] int x; // Ok, closed constructed type  
 [A(typeof(List<>))] int y; // Ok, unbound generic type  
}

## Attribute instances

An attribute instance is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

### Compilation of an attribute

The compilation of an attribute with attribute class T, positional-argument-list P and named-argument-list N, consists of the following steps:

* Follow the compile-time processing steps for compiling an object-creation-expression of the form new T(P). These steps either result in a compile-time error, or determine an instance constructor C on T that can be invoked at 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 must 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.
* Keep the following information for run-time instantiation of the attribute: the attribute class T, the instance constructor C on T, the positional-argument-list P and the named-argument-list N.

### Run-time retrieval of an attribute instance

Compilation of an attribute yields an attribute class T, an instance constructor C on T, a positional-argument-list P, and a named-argument-list N. Given this information, an attribute instance can be retrieved at run-time using the following steps:

* Follow the run-time processing steps for executing an object-creation-expression of the form new T(P), using the instance constructor C as determined at compile-time. These steps either result in an exception, or produce an instance O of T.
* For each named-argument Arg in N, in order:
* Let Name be the identifier of the named-argument Arg. If Name does not identify a non-static public read-write field or property on O, then an exception is thrown.
* Let Value be the result of evaluating the attribute-argument-expression of Arg.
* If Name identifies a field on O, then set this field to Value.
* Otherwise, Name identifies a property on O. Set this property to Value.
* The result is O, an instance of the attribute class T that has been initialized with the positional-argument-list P and the named-argument-list N.

## Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

* System.AttributeUsageAttribute (§17.4.1), which is used to describe the ways in which an attribute class can be used.
* System.Diagnostics.ConditionalAttribute (§17.4.2), which is used to define conditional methods.
* System.ObsoleteAttribute (§17.4.3), which is used to mark a member as obsolete.
* System.Runtime.CompilerServices.CallerLineNumberAttribute, System.Runtime.CompilerServices.CallerFilePathAttribute and System.Runtime.CompilerServices.CallerMemberNameAttribute (§17.4.4), which are used to supply information about the calling context to optional parameters.

### The AttributeUsage attribute

The attribute AttributeUsage is used to describe the manner in which the attribute class can be used.

A class that is decorated with the AttributeUsage attribute must derive from System.Attribute, either directly or indirectly. Otherwise, a compile-time error occurs.

namespace System  
{  
 [AttributeUsage(AttributeTargets.Class)]  
 public class AttributeUsageAttribute: Attribute  
 {  
 public AttributeUsageAttribute(AttributeTargets validOn) {...}

public virtual bool AllowMultiple { get {...} set {...} }

public virtual bool Inherited { get {...} set {...} }

public virtual AttributeTargets ValidOn { get {...} }  
 }

public enum AttributeTargets  
 {  
 Assembly = 0x0001,  
 Module = 0x0002,  
 Class = 0x0004,  
 Struct = 0x0008,  
 Enum = 0x0010,  
 Constructor = 0x0020,  
 Method = 0x0040,  
 Property = 0x0080,  
 Field = 0x0100,  
 Event = 0x0200,  
 Interface = 0x0400,  
 Parameter = 0x0800,  
 Delegate = 0x1000,  
 ReturnValue = 0x2000,

All = Assembly | Module | Class | Struct | Enum | Constructor |   
 Method | Property | Field | Event | Interface | Parameter |   
 Delegate | ReturnValue  
 }  
}

### The Conditional attribute

The attribute Conditional enables the definition of conditional methods and conditional attribute classes.

namespace System.Diagnostics  
{  
 [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,  
 AllowMultiple = true)]  
 public class ConditionalAttribute: Attribute  
 {  
 public ConditionalAttribute(string conditionString) {...}

public string ConditionString { get {...} }  
 }  
}

#### Conditional methods

A method decorated with the Conditional attribute is a conditional method. The Conditional attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the receiver and parameters of the call) is omitted.

A conditional method is subject to the following restrictions:

* The conditional method must be a method in a class-declaration or struct-declaration. A compile-time error occurs if the Conditional attribute is specified on a method in an interface declaration.
* The conditional method must have a return type of void.
* The conditional method must not be marked with the override modifier. A conditional method may be marked with the virtual modifier, however. Overrides of such a method are implicitly conditional, and must not be explicitly marked with a Conditional attribute.
* The conditional method must not be an implementation of an interface method. Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional method is used in a delegate-creation-expression. The example

#define DEBUG

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void M() {  
 Console.WriteLine("Executed Class1.M");  
 }  
}

class Class2  
{  
 public static void Test() {  
 Class1.M();  
 }  
}

declares Class1.M as a conditional method. Class2's Test method calls this method. Since the conditional compilation symbol DEBUG is defined, if Class2.Test is called, it will call M. If the symbol DEBUG had not been defined, then Class2.Test would not call Class1.M.

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. In the example

File class1.cs:

using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void F() {  
 Console.WriteLine("Executed Class1.F");  
 }  
}

File class2.cs:

#define DEBUG

class Class2  
{  
 public static void G() {  
 Class1.F(); // F is called  
 }  
}

File class3.cs:

#undef DEBUG

class Class3  
{  
 public static void H() {  
 Class1.F(); // F is not called  
 }  
}

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.

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through base, of the form base.M, are subject to the normal conditional method call rules. In the example

File class1.cs:

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public virtual void M() {  
 Console.WriteLine("Class1.M executed");  
 }  
}

File class2.cs:

using System;

class Class2: Class1  
{  
 public override void M() {  
 Console.WriteLine("Class2.M executed");  
 base.M(); // base.M is not called!  
 }  
}

File class3.cs:

#define DEBUG

using System;

class Class3  
{  
 public static void Test() {  
 Class2 c = new Class2();  
 c.M(); // M is called  
 }  
}

Class2 includes a call to the M defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol DEBUG, which is undefined. Thus, the method writes to the console “Class2.M executed” only. Judicious use of pp-declarations can eliminate such problems.

#### Conditional attribute classes

An attribute class (§17.1) decorated with one or more Conditional attributes is a conditional attribute class. A conditional attribute class is thus associated with the conditional compilation symbols declared in its Conditional attributes. This example:

using System;  
using System.Diagnostics;  
[Conditional("ALPHA")]  
[Conditional("BETA")]  
public class TestAttribute : Attribute {}

declares TestAttribute as a conditional attribute class associated with the conditional compilations symbols ALPHA and BETA.

Attribute specifications (§17.2) of a conditional attribute are included if one or more of its associated conditional compilation symbols 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. 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.

### The Obsolete attribute

The attribute Obsolete is used to mark types and members of types that should no longer be used.

namespace System  
{  
 [AttributeUsage(  
 AttributeTargets.Class |   
 AttributeTargets.Struct |  
 AttributeTargets.Enum |   
 AttributeTargets.Interface |   
 AttributeTargets.Delegate |  
 AttributeTargets.Method |   
 AttributeTargets.Constructor |  
 AttributeTargets.Property |   
 AttributeTargets.Field |  
 AttributeTargets.Event,  
 Inherited = false)  
 ]  
 public class ObsoleteAttribute: Attribute  
 {  
 public ObsoleteAttribute() {...}

public ObsoleteAttribute(string message) {...}

public ObsoleteAttribute(string message, bool error) {...}

public string Message { get {...} }

public bool IsError { get {...} }  
 }  
}

If a program uses a type or member that is decorated with the Obsolete attribute, the compiler issues a warning or an error. Specifically, the compiler issues a warning if no error parameter is provided, or if the error parameter is provided and has the value false. The compiler issues an error if the error parameter is specified and has the value true.

In the example

[Obsolete("This class is obsolete; use class B instead")]  
class A  
{  
 public void F() {}  
}

class B  
{  
 public void F() {}  
}

class Test  
{  
 static void Main() {  
 A a = new A(); // Warning  
 a.F();  
 }  
}

the class A is decorated with the Obsolete attribute. Each use of A in Main results in a warning that includes the specified message, “This class is obsolete; use class B instead.”

### Caller info attributes

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.

For 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);  
}

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.

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 (§7.5.3).

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.

#### The CallerLineNumber attribute

The System.Runtime.CompilerServices.CallerLineNumberAttribute is allowed on optional parameters when there is a standard implicit conversion (§6.3.1) 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.

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.

Note that the line number may be affected by #line directives (§2.5.7).

#### The CallerFilePath attribute

The System.Runtime.CompilerServices.CallerFilePathAttribute is allowed on optional parameters when there is a standard implicit conversion (§6.3.1) from string to the parameter’s type.

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.

Note that the file path may be affected by #line directives (§2.5.7).

#### The CallerMemberName attribute

The System.Runtime.CompilerServices.CallerMemberNameAttribute is allowed on optional parameters when there is a standard implicit conversion (§6.3.1) from string to the parameter’s type.

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 (§17.5.2.1) on the indexer member, if present, or the default name Item otherwise.

For invocations that occur within declarations of instance constructors, static constructors, destructors and operators the member name used is implementation-dependent.

## Attributes for Interoperation

Note: This section is applicable only to the Microsoft .NET implementation of C#.

### Interoperation with COM and Win32 components

The .NET run-time provides a large number of attributes that enable C# programs to interoperate with components written using COM and Win32 DLLs. For example, the DllImport attribute can be used on a static extern method to indicate that the implementation of the method is to be found in a Win32 DLL. These attributes are found in the System.Runtime.InteropServices namespace, and detailed documentation for these attributes is found in the .NET runtime documentation.

### Interoperation with other .NET languages

#### The IndexerName attribute

Indexers are implemented in .NET using indexed properties, and have a name in the .NET metadata. If no IndexerName attribute is present for an indexer, then the name Item is used by default. The IndexerName attribute enables a developer to override this default and specify a different name.

namespace System.Runtime.CompilerServices.CSharp  
{  
 [AttributeUsage(AttributeTargets.Property)]  
 public class IndexerNameAttribute: Attribute  
 {  
 public IndexerNameAttribute(string indexerName) {...}

public string Value { get {...} }   
 }  
}

23

# extern code

24

# INTERRUPT HANDLING

25

# THREADS AND LOCKS

26

# type inference

27

# INTERFACING TO OTHER LANGUAGES

28

# APPLICATION BINARY INTERFACE

# REFERENCES