L2/02-041

Standard ECMA-334
December 2001

ECMA

Standardizing Information and Communication Systems

C# Language Specification

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

Brief history

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

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

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

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

The development of this standard started in November 2000.

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

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

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

Table of Contents

1. Scope

This clause is informative.

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

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

This ECMA Standard does not specify

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

End of informative text.

2. Conformance

Conformance is of interest to the following audiences:

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

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

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

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

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

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

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

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

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

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

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

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

A conforming implementation of C# shall support the class library documented in §D. This library is included by reference in this ECMA Standard.

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

3. References

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

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

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

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

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

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

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

The following references are informative:

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

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

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

End of informative references

4. Definitions

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

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

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

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

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

Behavior — external appearance or action.

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

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

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

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

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

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

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

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

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

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

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

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

Program instantiation — the execution of an application.
Recommended practice — specification that is strongly recommended as being aligned with the intent of the standard, but that may be impractical for some implementations

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

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

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

5. Notational conventions

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

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

class-modifier:
  `new`
  `public`
  `protected`
  `internal`
  `private`
  `abstract`
  `sealed`

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

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

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

is equivalent to:

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

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

for-statement:
  `for` `(` *for-initializer*$_{opt}$ `;` *for-condition*$_{opt}$ `;` *for-iterator*$_{opt}$ `)` *embedded-statement*

is equivalent to:

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

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

6. Acronyms and abbreviations

This clause is informative.

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

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

CLI — Common Language Infrastructure

CLS — Common Language Specification

IEC — the International Electrotechnical Commission

IEEE — the Institute of Electrical and Electronics Engineers

ISO — the International Organization for Standardization

The name C# is pronounced "C Sharp".

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

End of informative text.

7. General description

This clause is informative.

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

This standard is divided into the following subdivisions:

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

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

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

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

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

End of informative text.

8. Language Overview

This clause is informative.

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

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

8.1 Getting started

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

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

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

```
csc hello.cs
```

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

```
hello, world
```

Close examination of this program is illuminating:

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

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

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

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

8.2 Types

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

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 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.

The example

```
using System;

class Class1
{
   public int Value = 0;
}

class Test
{
   static void Main() {
      int val1 = 0;
      int val2 = val1;
      val2 = 123;

      Class1 ref1 = new Class1();
      Class1 ref2 = ref1;
      ref2.Value = 123;

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

shows this difference. The output produced is

```
Values: 0, 123
Refs: 123, 123
```

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

The lines

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

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

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

```
using System;
public enum Color
{
   Red, Blue, Green
}
public struct Point
{
   public int x, y;
}
public interface IBase
{
   void F();
}
public interface IDerived: IBase
{
   void G();
}
public class A
{
   protected virtual void H() {
      Console.WriteLine("A.H");
   }
}
public class B: A, IDerived
{
   public void F() {
      Console.WriteLine("B.F, implementation of IDerived.F");
   }
   public void G() {
      Console.WriteLine("B.G, implementation of IDerived.G");
   }
   override protected void H() {
      Console.WriteLine("B.H, override of A.H");
   }
}
public delegate void EmptyDelegate();
```

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

8.2.1 Predefined types

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

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

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

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

```
int i = …;
F(i);
if (i = 0)  // Bug: the test should be (i == 0)
  G();
```

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

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

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

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

Type	**Description**	**Example**
`object`	The ultimate base type of all other types	`object o = null;`
`string`	String type; a string is a sequence of Unicode characters	`string s = "hello";`
`sbyte`	8-bit signed integral type	`sbyte val = 12;`
`short`	16-bit signed integral type	`short val = 12;`
`int`	32-bit signed integral type	`int val = 12;`
`long`	64-bit signed integral type	`long val1 = 12;` `long val2 = 34L;`
`byte`	8-bit unsigned integral type	`byte val1 = 12;`
`ushort`	16-bit unsigned integral type	`ushort val1 = 12;`
`uint`	32-bit unsigned integral type	`uint val1 = 12;` `uint val2 = 34U;`

`ulong`	64-bit unsigned integral type	`ulong val1 = 12;`
		`ulong val2 = 34U;`
		`ulong val3 = 56L;`
		`ulong val4 = 78UL;`

`float`	Single-precision floating point type	`float val = 1.23F;`
`double`	Double-precision floating point type	`double val1 = 1.23;` `double val2 = 4.56D;`
`bool`	Boolean type; a `bool` value is either true or false	`bool val1 = true;` `bool val2 = false;`
`char`	Character type; a `char` value is a Unicode character	`char val = 'h';`
`decimal`	Precise decimal type with 28 significant digits	`decimal val = 1.23M;`

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

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

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

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

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

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 == (object)t);
   }
}
```

produces the output

```
True
False
```

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

8.2.2 Conversions

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

```
using System;
class Test
{
   static void Main() {
      int intValue = 123;
      long longValue = intValue;
      Console.WriteLine("{0}, {1}", intValue, longValue);
   }
}
```

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

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

```
using System;
class Test
{
   static void Main() {
      long longValue = Int64.MaxValue;
      int intValue = (int) longValue;
      Console.WriteLine("(int) {0} = {1}", longValue, intValue);
   }
}
```

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

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

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

8.2.3 Array types

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

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

```
using System;
class Test
{
   static void Main() {
      int[] arr = new int[5];
      for (int i = 0; i < arr.Length; i++)
         arr[i] = i * i;
      for (int i = 0; i < arr.Length; i++)
         Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
   }
}
```

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

```
arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16
```

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

```
class Test
{
   static void Main() {
      int[] a1;          // single-dimensional array of int
      int[,] a2;         // 2-dimensional array of int
      int[,,] a3;        // 3-dimensional array of int
      int[][] j2;        // "jagged" array: array of (array of int)
      int[][][] j3;      // array of (array of (array of int))
   }
}
```

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

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

```
class Test
{
   static void Main() {
      int[] a1 = new int[] {1, 2, 3};
      int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
      int[,,] a3 = new int[10, 20, 30];
      int[][] j2 = new int[3][];
      j2[0] = new int[] {1, 2, 3};
      j2[1] = new int[] {1, 2, 3, 4, 5, 6};
      j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
   }
}
```

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

In contrast, the variable `j2` denotes a "jagged" array, or an "array of arrays". Specifically, `j2` denotes an array of an array of `int`, or a single-dimensional array of type `int[]`. Each of these `int[]` variables can be initialized individually, and this allows the array to take on a jagged shape. The example gives each of the `int[]` arrays a different length. Specifically, the length of `j2[0]` is 3, the length of `j2[1]` is 6, and the length of `j2[2]` is 9.

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

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

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

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

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

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

can be shortened to

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

without any change in program semantics.

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

```
class Test
{
   static void Main() {
      short[] a = {1, 2, 3};
      int[] b = {1, 2, 3};
      long[] c = {1, 2, 3};

   }
}
```

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

8.2.4 Type system unification

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

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine(3.ToString());
   }
}
```

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

The example

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

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

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

```
public class Stack
{
   public object Pop() {…}
   public void Push(object o) {…}
}
```

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

8.3 Variables and parameters

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

```
int a;
int b = 1;
```

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

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

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

```
class Test
{
   static void Main() {
      int a;
      int b = 1;
      int c = a + b; // error, a not yet assigned
      …
   }
}
```

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

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

```
using Personnel.Data;
class Employee
{
   private static DataSet ds;
   public string Name;
   public decimal Salary;
   …
}
```

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

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

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

```
using System;
class Test {
   static void F(int p) {
      Console.WriteLine("p = {0}", p);
      p++;
   }

   static void Main() {
      int a = 1;
      Console.WriteLine("pre:  a = {0}", a);
      F(a);
      Console.WriteLine("post: a = {0}", a);
   }
}
```

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

```
pre:  a = 1
p = 1
post: a = 1
```

even though the value parameter `p` is modified.

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

```
using System;
class Test {
   static void Swap(ref int a, ref int b) {
      int t = a;
      a = b;
      b = t;
   }

   static void Main() {
      int x = 1;
      int y = 2;

      Console.WriteLine("pre:  x = {0}, y = {1}", x, y);
      Swap(ref x, ref y);
      Console.WriteLine("post: x = {0}, y = {1}", x, y);
   }
}
```

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

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

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

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

```
using System;
class Test {
   static void Divide(int a, int b, out int result, out int remainder) {
      result = a / b;
      remainder = a % b;
   }

   static void Main() {
      for (int i = 1; i < 10; i++)
         for (int j = 1; j < 10; j++) {
            int ans, r;
            Divide(i, j, out ans, out r);
            Console.WriteLine("{0} / {1} = {2}r{3}", i, j, ans, r);
         }
   }
}
```

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

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

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

```
using System;
class Test
{
   static void F(params int[] args) {
      Console.WriteLine("# of arguments: {0}", args.Length);
      for (int i = 0; i < args.Length; i++)
         Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
   }

   static void Main() {
      F();
      F(1);
      F(1, 2);
      F(1, 2, 3);
      F(new int[] {1, 2, 3, 4});
   }
}
```

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

```
# of arguments: 0
# of arguments: 1
   args[0] = 1
# of arguments: 2
   args[0] = 1
   args[1] = 2
# of arguments: 3
   args[0] = 1
   args[1] = 2
   args[2] = 3
# of arguments: 4
   args[0] = 1
   args[1] = 2
   args[2] = 3
   args[3] = 4
```

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

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

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

```
namespace System
{
   public class Console
   {
      public static void WriteLine(string s) {…}
      public static void WriteLine(string s, object a) {…}
      public static void WriteLine(string s, object a, object b) {…}
      …
      public static void WriteLine(string s, params object[] args) {…}
   }
}
```

8.4 Automatic memory management

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

The example

```
using System;
public class Stack
{
   private Node first = null;
   public bool Empty {
      get {
         return (first == null);
      }
   }
   public object Pop() {
      if (first == null)
         throw new Exception("Can't Pop from an empty Stack.");
      else {
         object temp = first.Value;
         first = first.Next;
         return temp;
      }
   }
   public void Push(object o) {
      first = new Node(o, first);
   }
   class Node
   {
      public Node Next;
      public object Value;
      public Node(object value): this(value, null) {}
      public Node(object value, Node next) {
         Next = next;
         Value = value;
      }
   }
}
```

shows a `Stack` class implemented as a linked list of `Node` instances. Node instances are created in the `Push` method and are garbage collected when no longer needed. A `Node` instance becomes eligible for garbage collection when it is no longer possible for any code to access it. For instance, when an item is removed from the `Stack`, the associated `Node` instance becomes eligible for garbage collection.

The example

```
class Test
{
   static void Main() {
      Stack s = new Stack();
      for (int i = 0; i < 10; i++)
         s.Push(i);
      s = null;
   }
}
```

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

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

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

The example

```
using System;
class Test
{
   static void WriteLocations(byte[] arr) {
      unsafe {
         fixed (byte* pArray = arr) {
            byte* pElem = pArray;
            for (int i = 0; i < arr.Length; i++) {
               byte value = *pElem;
               Console.WriteLine("arr[{0}] at 0x{1:X} is {2}",
                  i, (uint)pElem, value);
               pElem++;
            }
         }
      }
   }

   static void Main() {
      byte[] arr = new byte[] {1, 2, 3, 4, 5};
      WriteLocations(arr);
   }
}
```

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

```
arr[0] at 0x8E0360 is 1
arr[1] at 0x8E0361 is 2
arr[2] at 0x8E0362 is 3
arr[3] at 0x8E0363 is 4
arr[4] at 0x8E0364 is 5
```

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

8.5 Expressions

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

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

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

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

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

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

8.6 Statements

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

Statement	**Example**

Statement lists and block statements	static void Main() {
	    F();
	    G();
	    {
	        H();
	        I();
	    }
	}

Labeled statements and `goto` statements	static void Main(string[] args) {
	    if (args.Length == 0)
	        goto done;
	    Console.WriteLine(args.Length);
	
	done:
	    Console.WriteLine("Done");
	}

Local constant declarations	static void Main() {
	    const float pi = 3.14f;
	    const int r = 123;
	    Console.WriteLine(pi * r * r);
	}

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

Expression statements	static int F(int a, int b) {
	    return a + b;
	}
	static void Main() {
	    F(1, 2); // Expression statement
	}

`if` statements	static void Main(string[] args) {
	    if (args.Length == 0)
	        Console.WriteLine("No args");
	    else
	        Console.WriteLine("Args");
	}

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

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

<table>
<tr><td><code>do</code> statements</td><td><pre>static void Main() {
    string s;
    do { s = Console.ReadLine(); }
    while (s != "Exit");
}</pre></td></tr>
<tr><td><code>for</code> statements</td><td><pre>static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++)
        Console.WriteLine(args[i]);
}</pre></td></tr>
<tr><td><code>foreach</code> statements</td><td><pre>static void Main(string[] args) {
    foreach (string s in args)
        Console.WriteLine(s);
}</pre></td></tr>
<tr><td><code>break</code> statements</td><td><pre>static void Main(string[] args) {
    int i = 0;
    while (true) {
        if (i == args.Length)
            break;
        Console.WriteLine(args[i++]);
    }
}</pre></td></tr>
<tr><td><code>continue</code> statements</td><td><pre>static void Main(string[] args) {
    int i = 0;
    while (true) {
       Console.WriteLine(args[i++]);
       if (i < args.Length)
            continue;
       break;
    }
}</pre></td></tr>
<tr><td><code>return</code> statements</td><td><pre>static int F(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(F(1, 2));
    return;
}</pre></td></tr>
<tr><td><code>throw</code> statements and <code>try</code> statements</td><td><pre>static int F(int a, int b) {
    if (b == 0)
        throw new Exception("Divide by zero");
    return a / b;
}

static void Main() {
    try {
        Console.WriteLine(F(5, 0));
    }
    catch(Exception e) {
        Console.WriteLine("Error");
    }
}</pre></td></tr>
<tr><td><code>checked</code> and <code>unchecked</code> statements</td><td><pre>static void Main() {
    int x = Int32.MaxValue;
    Console.WriteLine(x + 1);   // Overflow
    checked {
        Console.WriteLine(x + 1); // Exception
    }
    unchecked {
       Console.WriteLine(x + 1); // Overflow
    }
}</pre></td></tr>
</table>

`lock` statements	<pre>static void Main() {
	    A a = …;
	    lock(a) {
	        a.P = a.P + 1;
	    }
	}</pre>

`using` statements	<pre>static void Main() {
	    using (Resource r = new Resource()) {
	        r.F();
	    }
	}</pre>

8.7 Classes

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

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

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

The example

```
using System;
class MyClass
{
   public MyClass() {
      Console.WriteLine("Instance constructor");
   }
   public MyClass(int value) {
      MyField = value;
      Console.WriteLine("Instance constructor");
   }
   ~MyClass() {
      Console.WriteLine("Destructor");
   }
   public const int MyConst = 12;
   public int MyField = 34;
   public void MyMethod(){
      Console.WriteLine("MyClass.MyMethod");
   }
   public int MyProperty {
      get {
         return MyField;
      }
```

```
      set {
         MyField = value;
      }
   }
   public int this[int index] {
      get {
         return 0;
      }
      set {
         Console.WriteLine("this[{0}] = {1}", index, value);
      }
   }
   public event EventHandler MyEvent;
   public static MyClass operator+(MyClass a, MyClass b) {
      return new MyClass(a.MyField + b.MyField);
   }
   internal class MyNestedClass
   {}
}
```

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

```
class Test
{
   static void Main() {
      // Instance constructor usage
      MyClass a = new MyClass();
      MyClass b = new MyClass(123);
      // Constant usage
      Console.WriteLine("MyConst = {0}", MyClass.MyConst);
      // Field usage
      a.MyField++;
      Console.WriteLine("a.MyField = {0}", a.MyField);
      // Method usage
      a.MyMethod();
      // Property usage
      a.MyProperty++;
      Console.WriteLine("a.MyProperty = {0}", a.MyProperty);
      // Indexer usage
      a[3] = a[1] = a[2];
      Console.WriteLine("a[3] = {0}", a[3]);
      // Event usage
      a.MyEvent += new EventHandler(MyHandler);
      // Overloaded operator usage
      MyClass c = a + b;
   }
   static void MyHandler(object sender, EventArgs e) {
      Console.WriteLine("Test.MyHandler");
   }
   internal class MyNestedClass
   {}
}
```

shows uses of these members.

8.7.1 Constants

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

```
class Constants
{
   public const int A = 1;
   public const int B = A + 1;
}
```

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

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

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine("{0}, {1}", Constants.A, Constants.B);
   }
}
```

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

8.7.2 Fields

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

```
class Color
{
   internal ushort redPart;
   internal ushort bluePart;
   internal ushort greenPart;

   public Color(ushort red, ushort blue, ushort green) {
      redPart = red;
      bluePart = blue;
      greenPart = green;
   }

   public static Color Red = new Color(0xFF, 0, 0);
   public static Color Blue = new Color(0, 0xFF, 0);
   public static Color Green = new Color(0, 0, 0xFF);
   public static Color White = new Color(0xFF, 0xFF, 0xFF);
}
```

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

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

```
class Color
{
   internal ushort redPart;
   internal ushort bluePart;
   internal ushort greenPart;

   public Color(ushort red, ushort blue, ushort green) {
      redPart = red;
      bluePart = blue;
      greenPart = green;
   }
```

```
    public static readonly Color Red = new Color(0xFF, 0, 0);
    public static readonly Color Blue = new Color(0, 0xFF, 0);
    public static readonly Color Green = new Color(0, 0, 0xFF);
    public static readonly Color White = new Color(0xFF, 0xFF, 0xFF);
}
```

8.7.3 Methods

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

```
using System;
public class Stack
{
   public static Stack Clone(Stack s) {…}

   public static Stack Flip(Stack s) {…}

   public object Pop() {…}

   public void Push(object o) {…}

   public override string ToString() {…}
   …
}
class Test
{
   static void Main() {
      Stack s = new Stack();
      for (int i = 1; i < 10; i++)
         s.Push(i);

      Stack flipped = Stack.Flip(s);

      Stack cloned = Stack.Clone(s);

      Console.WriteLine("Original stack: " + s.ToString());
      Console.WriteLine("Flipped stack: " + flipped.ToString());
      Console.WriteLine("Cloned stack: " + cloned.ToString());
   }
}
```

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

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

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

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

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

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

```
      static void F(int a, int b) {
         Console.WriteLine("F(int, int)");
      }
      static void F(int[] values) {
         Console.WriteLine("F(int[])");
      }
      static void Main() {
         F();
         F(1);
         int i = 10;
         F(ref i);
         F((object)1);
         F(1, 2);
         F(new int[] {1, 2, 3});
      }
   }
```

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

```
F()
F(int)
F(ref int)
F(object)
F(int, int)
F(int[])
```

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

```
public class Button
{
   private string caption;
   public string Caption {
      get {
         return caption;
      }
      set {
         caption = value;
         Repaint();
      }
   }
}
```

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

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

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

8.7.5 Events

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

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

In the example

```
public delegate void EventHandler(object sender, System.EventArgs e);

public class Button
{
   public event EventHandler Click;
   public void Reset() {
      Click = null;
   }
}
```

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

```
using System;

public class Form1
{
   public Form1() {
      // Add Button1_Click as an event handler for Button1's Click event
      Button1.Click += new EventHandler(Button1_Click);
   }

   Button Button1 = new Button();

   void Button1_Click(object sender, EventArgs e) {
      Console.WriteLine("Button1 was clicked!");
   }

   public void Disconnect() {
      Button1.Click -= new EventHandler(Button1_Click);
   }
}
```

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

For a simple event declaration such as

```
public event EventHandler Click;
```

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

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

```
public class Button
{
      private EventHandler handler;
      public event EventHandler Click {
         add { handler += value; }
```

```
            remove { handler -= value; }
        }
    }
```

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

8.7.6 Operators

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

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

```
using System;
public struct Digit
{
   byte value;
   public Digit(byte value) {
      if (value < 0 || value > 9) throw new ArgumentException();
      this.value = value;
   }
   public Digit(int value): this((byte) value) {}
   public static implicit operator byte(Digit d) {
      return d.value;
   }
   public static explicit operator Digit(byte b) {
      return new Digit(b);
   }
   public static Digit operator+(Digit a, Digit b) {
      return new Digit(a.value + b.value);
   }
   public static Digit operator-(Digit a, Digit b) {
      return new Digit(a.value - b.value);
   }
   public static bool operator==(Digit a, Digit b) {
      return a.value == b.value;
   }
   public static bool operator!=(Digit a, Digit b) {
      return a.value != b.value;
   }
   public override bool Equals(object value) {
      if (value == null) return false;
      if (GetType() == value.GetType()) return this == (Digit)value;
      return false;  }
   public override int GetHashCode() {
      return value.GetHashCode();
   }
   public override string ToString() {
      return value.ToString();
   }
}
```

```
class Test
{
   static void Main() {
      Digit a = (Digit) 5;
      Digit b = (Digit) 3;
      Digit plus = a + b;
      Digit minus = a - b;
      bool equals = (a == b);
      Console.WriteLine("{0} + {1} = {2}", a, b, plus);
      Console.WriteLine("{0} - {1} = {2}", a, b, minus);
      Console.WriteLine("{0} == {1} = {2}", a, b, equals);
   }
}
```

The `Digit` type defines the following operators:

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

8.7.7 Indexers

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

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

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

```
using System;
public class Stack
{
   private Node GetNode(int index) {
      Node temp = first;
      while (index > 0) {
         temp = temp.Next;
         index--;
      }
      return temp;
   }

   public object this[int index] {
      get {
         if (!ValidIndex(index))
            throw new Exception("Index out of range.");
         else
            return GetNode(index).Value;
      }

      set {
         if (!ValidIndex(index))
            throw new Exception("Index out of range.");
         else
            GetNode(index).Value = value;
      }
   }

   …
}
```

```
class Test
{
   static void Main() {
      Stack s = new Stack();

      s.Push(1);
      s.Push(2);
      s.Push(3);

      s[0] = 33;  // Changes the top item from 3 to 33
      s[1] = 22;  // Changes the middle item from 2 to 22
      s[2] = 11;  // Changes the bottom item from 1 to 11
   }
}
```

shows an indexer for the `Stack` class.

8.7.8 Instance constructors

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

The example

```
using System;
class Point
{
   public double x, y;

   public Point() {
      this.x = 0;
      this.y = 0;
   }

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

   public static double Distance(Point a, Point b) {
      double xdiff = a.x - b.x;
      double ydiff = a.y - b.y;
      return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
   }

   public override string ToString() {
      return string.Format("({0}, {1})", x, y);
   }
}

class Test
{
   static void Main() {
      Point a = new Point();
      Point b = new Point(3, 4);
      double d = Point.Distance(a, b);
      Console.WriteLine("Distance from {0} to {1} is {2}", a, b, d);
   }
}
```

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

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

8.7.9 Destructors

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

The example

```
using System;
class Point
{
   public double x, y;
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   ~Point() {
      Console.WriteLine("Destructed {0}", this);
   }
   public override string ToString() {
      return string.Format("({0}, {1})", x, y);
   }
}
```

shows a `Point` class with a destructor.

8.7.10 Static constructors

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

The example

```
using Personnel.Data;
class Employee
{
   private static DataSet ds;
   static Employee() {
      ds = new DataSet(…);
   }
   public string Name;
   public decimal Salary;
   …
}
```

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

8.7.11 Inheritance

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

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

```
using System;
class A
{
   public void F() { Console.WriteLine("A.F"); }
}
```

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

```
class B: A
{
   public void G() { Console.WriteLine("B.G"); }
}
class Test
{
   static void Main() {
      B b = new B();
      b.F();         // Inherited from A
      b.G();         // Introduced in B
```

```
        A a = b;        // Treat a B as an A
        a.F();
    }
}
```

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

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

```
using System;
class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
    public override void F() {
        base.F();
        Console.WriteLine("B.F");
    }
}
class Test
{
    static void Main() {
        B b = new B();
        b.F();
        A a = b;
        a.F();
    }
}
```

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

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

```
using System;
abstract class A
{
    public abstract void F();
}
class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}
class Test
{
    static void Main() {
        B b = new B();
        b.F();
        A a = b;
        a.F();
    }
}
```

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

8.8 Structs

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

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

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

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

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

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

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

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

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

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

8.9 Interfaces

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

The example

```
interface IExample
{
   string this[int index] { get; set; }
   event EventHandler E;
   void F(int value);
   string P { get; set; }
}

public delegate void EventHandler(object sender, EventArgs e);
```

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

Interfaces may employ multiple inheritance. 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 interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

Classes and structs can implement multiple interfaces. In the example

```
interface IDataBound
{
   void Bind(Binder b);
}
public class EditBox: Control, IControl, IDataBound
{
   public void Paint() {…}
   public void Bind(Binder b) {…}
}
```

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

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

```
public class EditBox: IControl, IDataBound
{
   void IControl.Paint() {…}
   void IDataBound.Bind(Binder b) {…}
}
```

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

```
class Test
{
   static void Main() {
      EditBox editbox = new EditBox();
      editbox.Paint();  // error: no such method
      IControl control = editbox;
      control.Paint();  // calls EditBox's Paint implementation
   }
}
```

8.10 Delegates

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

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates 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. Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate instance's methods with that set of arguments.

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 (§22.1) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation. This is a powerful capability.

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

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

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

The example

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

   static void Main() {
      SimpleDelegate d = new SimpleDelegate(F);
      d();
   }
}
```

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

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

```
void MultiCall(SimpleDelegate d, int count) {
   for (int i = 0; i < count; i++)
      d();
   }
}
```

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

8.11 Enums

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

The example

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

class Shape
{
   public void Fill(Color color) {
      switch(color) {
         case Color.Red:
            …
            break;

         case Color.Blue:
            …
            break;
```

```
            case Color.Green:
               …
               break;
            default:
               break;
         }
      }
   }
```

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

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

8.12 Namespaces and assemblies

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

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

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

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

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

```
// HelloLibrary.cs
namespace CSharp.Introduction
{
   public class HelloMessage
   {
      public string Message {
         get {
            return "hello, world";
         }
      }
   }
}
```

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

```
namespace CSharp.Introduction
{…}
```

is shorthand for two levels of namespace nesting:

```
namespace CSharp
{
   namespace Introduction
   {…}
}
```

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

```
// HelloApp.cs

using CSharp.Introduction;

class HelloApp
{
   static void Main() {
      HelloMessage m = new HelloMessage();
      System.Console.WriteLine(m.Message);
   }
}
```

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

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

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

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

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

```
csc /target:library HelloLibrary.cs

csc /reference:HelloLibrary.dll HelloApp.cs
```

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

8.13 Versioning

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

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

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

```
// Author A
namespace A
{
   public class Base    // version 1
   {
   }
}
```

```
// Author B
namespace B
{
   class Derived: A.Base
   {
      public virtual void F() {
         System.Console.WriteLine("Derived.F");
      }
   }
}
```

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

```
// Author A
namespace A
{
   public class Base    // version 2
   {
      public virtual void F() {     // added in version 2
         System.Console.WriteLine("Base.F");
      }
   }
}
```

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

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

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

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

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

```
// Author A
namespace A
{
   public class Base       // version 2
   {
      public virtual void F() { // added in version 2
         System.Console.WriteLine("Base.F");
      }
   }
}
```

```
// Author B
namespace B
{
   class Derived: A.Base   // version 2a: new
   {
      new public virtual void F() {
         System.Console.WriteLine("Derived.F");
      }
   }
}
```

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

```
// Author A
namespace A
{
   public class Base             // version 2
   {
      public virtual void F() { // added in version 2
         System.Console.WriteLine("Base.F");
      }
   }
}
// Author B
namespace B
{
   class Derived: A.Base   // version 2b: override
   {
      public override void F() {
         base.F();
         System.Console.WriteLine("Derived.F");
      }
   }
}
```

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

8.14 Attributes

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

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

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute: Attribute
{
   public HelpAttribute(string url) {
      this.url = url;
   }
   public string Topic = null;
   private string url;
```

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

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

The example

```
[Help("http://www.mycompany.com/…/Class1.htm")]
public class Class1
{
   [Help("http://www.mycompany.com/…/Class1.htm", Topic = "F")]
   public void F() {}
}
```

shows several uses of the attribute `Help`.

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

```
using System;
class Test
{
   static void Main() {
      Type type = typeof(Class1);
      object[] arr = type.GetCustomAttributes(typeof(HelpAttribute),
true);
      if (arr.Length == 0)
         Console.WriteLine("Class1 has no Help attribute.");
      else {
         HelpAttribute ha = (HelpAttribute) arr[0];
         Console.WriteLine("Url = {0}, Topic = {1}", ha.Url, ha.Topic);
      }
   }
}
```

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

End of informative text.

9. Lexical structure

9.1 Programs

A C# ***program*** consists of one or more source files, known formally as ***compilation units*** (§16.1). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required.

Conceptually speaking, a program is compiled using three steps:

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

Conforming implementations must accept Unicode source files encoded with the UTF-8 encoding form (as defined by the Unicode standard), and transform them into a sequence of Unicode characters. Implementations may choose to accept and transform additional character encoding schemes (such as UTF-16, UTF-32, or non-Unicode character mappings).

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

9.2 Grammars

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

9.2.1 Lexical grammar

The lexical grammar of C# is presented in §9.3, §9.4, and §9.5. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form tokens (§9.4), white space (§9.3.3), comments (§9.3.2), and pre-processing directives (§9.5).

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

9.2.2 Syntactic grammar

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

Every source file in a C# program must conform to the *compilation-unit* production (§16.1) of the syntactic grammar.

9.3 Lexical analysis

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

input::
  input-section$_{opt}$

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

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

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

input-element::
  whitespace
  comment
  token

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

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

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

9.3.1 Line terminators

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

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

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

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

9.3.2 Comments

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

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

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

includes a delimited comment. *end example*]

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

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

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

comment::
  single-line-comment
  delimited-comment

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

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

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

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

delimited-comment::
  `/*` *delimited-comment-characters*$_{opt}$ `*/`

delimited-comment-characters::
  delimited-comment-character
  delimited-comment-characters *delimited-comment-character*

delimited-comment-character::
  not-asterisk
  `*` *not-slash*

not-asterisk::
  Any Unicode character except `*`

not-slash::
  Any Unicode character except `/`

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

Comments are not processed within character and string literals.

9.3.3 White space

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

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

9.4 Tokens

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

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

9.4.1 Unicode escape sequences

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

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

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

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

[*Example:* The example

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

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

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

end example]

9.4.2 Identifiers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[*Example:* The example:

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

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

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

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

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

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

9.4.3 Keywords

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

keyword:: one of

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

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

9.4.4 Literals

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

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

9.4.4.1 Boolean literals

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

boolean-literal::
  `true`
  `false`

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

9.4.4.2 Integer literals

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

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

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

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

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

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

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

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

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

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.

[*Note:* 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`". *end note*]

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

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

9.4.4.3 Real literals

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

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

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

sign:: one of
`+ -`

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

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

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

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

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

9.4.4.4 Character literals

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

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

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

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

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

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

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

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

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

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

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

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`.

9.4.4.5 String literals

C# supports two forms of string literals: ***regular string literals*** and ***verbatim string literals***. A regular string literal consists of zero or more characters enclosed in double quotes, as in `"hello, world"`, 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, world"`. In a verbatim string literal, the characters between the delimiters are interpreted verbatim, with 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.

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

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

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

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

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

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

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

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

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

quote-escape-sequence::
  `""`

[*Note:* 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. *end note*]

[*Example:* The example

```
string a = "Happy birthday, Joel";        // Happy birthday, Joel
string b = @"Happy birthday, Joel";       // Happy birthday, Joel

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

string e = "Joe said \"Hello\" to me";    // 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. *end example*]

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

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

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

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

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

9.4.4.6 The null literal

null-literal::
  `null`

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

9.4.5 Operators and punctuators

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

operator-or-punctuator:: one of

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

9.5 Pre-processing directives

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

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

The following pre-processing directives are available:

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

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

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

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

```
#define A
#undef B

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

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

results in the exact same sequence of tokens as the program

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

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

9.5.1 Conditional compilation symbols

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

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

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

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

9.5.2 Pre-processing expressions

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

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

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

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

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

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

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

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

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

9.5.3 Declaration directives

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

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

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

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

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

[*Example:* The example:

```
#define Enterprise

#if Professional || Enterprise
   #define Advanced
#endif

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

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

end example]

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

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

end example]

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

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

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

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

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

end example]

9.5.4 Conditional compilation directives

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

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

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

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

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

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

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

conditional-section::
input-section
skipped-section

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#define Debug     // Debugging on

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

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

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

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

results in the output:

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

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

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

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

9.5.5 Diagnostic directives

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

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

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

[*Example:* The example

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

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

class Test {…}
```

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

9.5.6 Region control

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

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

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

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

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

The lexical processing of a region:

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

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

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

9.5.7 Line directives

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

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

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

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

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

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

file-name-character::
Any character except `"` (`U+0022`), and *new-line*

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

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

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

10. Basic concepts

10.1 Application 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 shall have one of the following signatures:

```
static void Main() {…}
static void Main(string[] args) {…}
static int Main() {…}
static int Main(string[] args) {…}
```

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

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

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

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

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

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

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

10.2 Application termination

Application termination returns control to the execution environment.

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

If the return type of the entry point method is `void`, reaching the right brace (`}`) 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, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

10.3 Declarations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end note]

10.4 Members

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

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

10.4.1 Namespace members

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

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

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

10.4.2 Struct members

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

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

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.

- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

10.4.3 Enumeration members

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

10.4.4 Class members

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

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

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

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

10.4.5 Interface members

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

10.4.6 Array members

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

10.4.7 Delegate members

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

10.5 Member access

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

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

10.5.1 Declared accessibility

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

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

- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program".
- Protected internal, which is selected by including both a `protected` and an `internal` modifier in the member declaration. The intuitive meaning of `protected internal` is "access limited to this program or types derived from the containing class".
- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type".

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

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

10.5.2 Accessibility domains

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

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

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

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

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

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

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

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

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

- First, if `M` is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if `M` is `public`, the access is permitted.
- Otherwise, if `M` is `protected internal`, the access is permitted if it occurs within the program in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (§10.5.3).
- Otherwise, if `M` is `protected`, the access is permitted if it occurs within the class in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (§10.5.3).
- Otherwise, if `M` is `internal`, the access is permitted if it occurs within the program in which `M` is declared.
- Otherwise, if `M` is `private`, the access is permitted if it occurs within the type in which `M` is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

end note]

[*Example:* In the example

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

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

the classes and members have the following accessibility domains:

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

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

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

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

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

10.5.3 Protected access for instance members

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

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

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

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

[*Example:* In the example

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

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

10.5.4 Accessibility constraints

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

The following accessibility constraints exist:

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

[*Example:* In the example

```
class A {…}
public class B: A {…}
```

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

[*Example:* Likewise, in the example

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

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

10.6 Signatures and overloading

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

- The signature of a method consists of the name of the method and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of a method specifically does not include the return type, nor does it include the `params` modifier that may be specified for the right-most parameter.
- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the right-most parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

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

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

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

```
interface ITest
{
   void F();                     // F()
   void F(int x);                // F(int)
   void F(ref int x);            // F(ref int)
   void F(out int x);            // F(out int)
   void F(int x, int y);         // F(int, int)
   int F(string s);              // F(string)
```

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

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

10.7 Scopes

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

- The scope of a namespace member declared by a *namespace-member-declaration* (§16.4) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is `N`, is the *namespace-body* of every *namespace-declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of a name defined or imported by a *using-directive* (§16.3) extends over the *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A *using-directive* may make zero or more namespace or type names available within a particular *compilation-unit* or *namespace-body*, but does not contribute any new members to the underlying declaration space. In other words, a *using-directive* is not transitive, but, rather, affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a member declared by a *class-member-declaration* (§17.2) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (§10.5.2) of the member.
- The scope of a member declared by a *struct-member-declaration* (§18.2) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§21.3) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§17.5) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§17.8) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§17.9) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§17.10) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a label declared in a *labeled-statement* (§15.4) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§15.5.1) is the block in which the declaration occurs.

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

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

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

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

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

```
class A
{
   int i = 0;

   void F() {
      i = 1;                 // Error, use precedes declaration
      int i;
      i = 2;
   }

   void G() {
      int j = (j = 1);       // Valid
   }

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

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

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

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

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

```
class Test
{
   static void Main() {
      string A = "hello, world";
      string s = A;                        // expression context

      Type t = typeof(A);                  // type context

      Console.WriteLine(s);                // writes "hello, world"
      Console.WriteLine(t.ToString());     // writes "Type: A"
   }
}
```

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

10.7.1 Name hiding

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

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

10.7.1.1 Hiding through nesting

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

```
class A
{
   int i = 0;

   void F() {
      int i = 1;
   }

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

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

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

```
class Outer
{
   static void F(int i) {}

   static void F(string s) {}

   class Inner
   {
      void G() {
         F(1);          // Invokes Outer.Inner.F
         F("Hello");    // Error
      }

      static void F(long l) {}
   }
}
```

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

10.7.1.2 Hiding through inheritance

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

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

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

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

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

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

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

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

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

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

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

```
class MoreDerived: Derived
{
   static void G() { F(); }         // Invokes Base.F
}
```

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

10.8 Namespace and type names

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

namespace-name:
  namespace-or-type-name

type-name:
  namespace-or-type-name

namespace-or-type-name:
  identifier
  namespace-or-type-name `.` *identifier*

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

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

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

- If the *namespace-or-type-name* consists of a single identifier:
  - If the *namespace-or-type-name* appears within the body of a class or struct declaration, then starting with that class or struct declaration and continuing with each enclosing class or struct declaration (if any), if a member with the given name exists, is accessible, and denotes a type, then the *namespace-or-type-name* refers to that member. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) are ignored when determining the meaning of a *namespace-or-type-name*.
  - Otherwise, starting with the namespace in which the *namespace-or-type-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
    - If the namespace contains a namespace member with the given name, then the *namespace-or-type-name* refers to that member and, depending on the member, is classified as a namespace or a type.
    - Otherwise, if the namespace has a corresponding namespace declaration enclosing the location where the *namespace-or-type-name* occurs, then:
      - If the namespace declaration contains a using-alias-directive that associates the given name with an imported namespace or type, then the namespace-or-type-name refers to that namespace or type.
      - Otherwise, if the namespaces imported by the using-namespace-directives of the namespace declaration contain exactly one type with the given name, then the namespace-or-type-name refers to that type.
      - Otherwise, if the namespaces imported by the using-namespace-directives of the namespace declaration contain more than one type with the given name, then the namespace-or-type-name is ambiguous and an error occurs.

- Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.

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

10.8.1 Fully qualified names

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

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

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

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

```
class A {}            // A
namespace X           // X
{
   class B            // X.B
   {
      class C {}      // X.B.C
   }
   namespace Y        // X.Y
   {
      class D {}      // X.Y.D
   }
}
namespace X.Y         // X.Y
{
   class E {}         // X.Y.E
}
```

end example]

10.9 Automatic memory management

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

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

execution from the current execution point in the procedure, an implementation may (but is not required to) treat the object as no longer in use. *end note*]

3. Once the object is eligible for destruction, at some unspecified later time the destructor (§17.12) (if any) for the object is run. Unless overridden by explicit calls, the destructor for the object is run once only.
4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered ***inaccessible*** and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

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

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in any particular order, or on any particular thread.

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

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

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

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

```
Destruct instance of A
Destruct instance of B
```

or

```
Destruct instance of B
Destruct instance of A
```

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

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

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

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

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

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

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

10.10 Execution order

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

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

11. Types

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

type:
    value-type
    reference-type

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

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

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

11.1 Value types

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

value-type:
    struct-type
    enum-type

struct-type:
    type-name
    simple-type

simple-type:
    numeric-type
    `bool`

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

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

floating-point-type:
`float`
`double`

enum-type:
type-name

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

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

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.

11.1.1 Default constructors

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

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

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

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

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

11.1.2 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. Struct types are described in §18.

11.1.3 Simple types

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

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

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

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

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

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

11.1.4 Integral types

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

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

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

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

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

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

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

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

11.1.5 Floating point types

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

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (§14.7.2).
- 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 $s \times m \times 2^e$, where s is 1 or −1, and m and e are determined by the particular floating-point type: For `float`, $0 < m < 2^{24}$ and $-149 \le e \le 104$, and for `double`, $0 < m < 2^{53}$ and $-1075 \le e \le 970$. Denormalized floating-point numbers are considered valid non-zero values.

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

The `double` type can represent values ranging from approximately 5.0×10^{-324} to 1.7×10^{308} with a precision of 15–16 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, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

11.1.6 The `decimal` type

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

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

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

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

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

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

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

11.1.7 The `bool` type

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

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

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

11.1.8 Enumeration types

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

11.2 Reference types

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

reference-type:
  class-type
  interface-type
  array-type
  delegate-type

class-type:
  type-name
  `object`
  `string`

interface-type:
  type-name

array-type:
  non-array-type rank-specifiers

non-array-type:
  type

rank-specifiers:
  rank-specifier
  rank-specifiers rank-specifier

rank-specifier:
  `[` *dim-separators*$_{opt}$ `]`

dim-separators:
  `,`
  dim-separators `,`

delegate-type:
  type-name

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

11.2.1 Class types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using *object-creation-expressions* (§14.5.10.1).

Class types are described in §17.

11.2.2 The `object` type

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

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

11.2.3 The `string` type

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

Values of the `string` type can be written as string literals (§9.4.4).

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

11.2.4 Interface types

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

Interface types are described in §20.

11.2.5 Array types

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

Array types are described in §19.

11.2.6 Delegate types

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

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

Delegate types are described in §22.

11.3 Boxing and unboxing

The concept of boxing and unboxing is central to C#'s type system. It provides a bridge between *value-type*s and *reference-type*s by permitting any value of a *value-type* to be converted to and from type `object`. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

11.3.1 Boxing conversions

A boxing conversion permits any *value-type* to be implicitly converted to the type `object` or to any *interface-type* implemented by the *value-type*. Boxing a value of a *value-type* consists of allocating an object instance and copying the *value-type* value into that instance.

The actual process of boxing a value of a *value-type* is best explained by imagining the existence of a ***boxing class*** for that type. [*Example:* For any *value-type* `T`, the boxing class behaves as if it were declared as follows:

```
sealed class T_Box
{
   T value;
   public T_Box(T t) {
      value = t;
   }
}
```

Boxing of a value `v` of type `T` now consists of executing the expression `new T_Box(v)`, and returning the resulting instance as a value of type `object`. Thus, the statements

```
int i = 123;
object box = i;
```

conceptually correspond to

```
int i = 123;
object box = new int_Box(i);
```

end example]

Boxing classes like `T_Box` and `int_Box` above don't actually exist and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type `T` has the dynamic type `T`, and a dynamic type check using the `is` operator can simply reference type `T`. [*Example:* For example,

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

will output the string "`Box contains an int`" on the console. *end example*]

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

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

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

the following statements

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

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance. *end example*]

11.3.2 Unboxing conversions

An unboxing conversion permits an explicit conversion from type `object` to any *value-type* or from any *interface-type* to any *value-type* that implements the *interface-type*. An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object `box` to a *value-type* `T` consists of executing the expression `((T_Box)box).value`. [*Example:* Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new int_Box(123);
int i = ((int_Box)box).value;
```

end example]

For an unboxing conversion to a given *value-type* to succeed at run-time, the value of the source operand must be a reference to an object that was previously created by boxing a value of that *value-type*. If the source operand is `null` or a reference to an incompatible object, a `System.InvalidCastException` is thrown.

12. Variables

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

A variable must be ***definitely assigned*** (§12.3) before its value can be obtained.

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

12.1 Variable categories

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

[*Example:* In the example

```
class A
{
   public static int x;
   int y;

   void F(int[] v, int a, ref int b, out int c) {
      int i = 1;
      c = a + b++;
   }
}
```

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

12.1.1 Static variables

A field declared with the `static` modifier is called a ***static variable***. A static variable comes into existence before execution of the static constructor (§17.11) 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 (§12.2) of the variable's type.

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

12.1.2 Instance variables

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

12.1.2.1 Instance variables in classes

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

The initial value of an instance variable of a class is the default value (§12.2) of the variable's type.

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

12.1.2.2 Instance variables in structs

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

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

12.1.3 Array elements

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

The initial value of each of the elements of an array is the default value (§12.2) of the type of the array elements.

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

12.1.4 Value parameters

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

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the function member.

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

12.1.5 Reference parameters

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

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

The following definite assignment rules apply to reference parameters. Note the different rules for output parameters described in §12.1.6.

- A variable must be definitely assigned (§12.3) before it can be passed as a reference parameter in a function member invocation.
- Within a function member, 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 (§14.5.7).

12.1.6 Output parameters

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

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

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters described in §12.1.5.

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member invocation.

- Following the normal completion of a function member invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member, an output parameter is considered initially unassigned.
- Every output parameter of a function member must be definitely assigned (§12.3) before the function member returns normally.

Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of the struct type (§14.5.7).

12.1.7 Local variables

A ***local variable*** is declared by a *local-variable-declaration*, which may occur in a *block*, a *for-statement*, a *switch-statement*, or a *using-statement*.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends from entry into the *block*, *for-statement*, *switch-statement*, or *using-statement* with which it is associated, until execution of that *block*, *for-statement*, *switch-statement*, or *using-statement* 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*, or *using-statement*.) If the parent *block*, *for-statement*, *switch-statement*, or *using-statement* 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 is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable is considered initially unassigned. A *local-variable-declaration* may include a *local-variable-initializer*, in which case the variable is considered definitely assigned in its entire scope, except within the expression provided in the *local-variable-initializer*.

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

[*Note:* The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is only used for a small portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable (§10.9). *end note*]

A local variable is also declared by a *foreach-statement* and by a *specific-catch-clause* for a *try-statement*. For a *foreach-statement*, the local variable is an iteration variable (§15.8.4). For a *specific-catch-clause*, the local variable is an exception variable (§15.10). A local variable declared by a *foreach-statement* or *specific-catch-clause* is considered definitely assigned in its entire scope.

12.2 Default values

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

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

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

- For a variable of a *value-type*, the default value is the same as the value computed by the *value-type*'s default constructor (§11.1.1).
- For a variable of a *reference-type*, the default value is `null`.

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

12.3 Definite assignment

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

- An initially assigned variable (§12.3.1) is always considered definitely assigned.
- An initially unassigned variable (§12.3.2) is considered definitely assigned at a given location if all possible execution paths leading to that location contain at least one of the following:
  - A simple assignment (§14.13.1) in which the variable is the left operand.
  - An invocation expression (§14.5.5) or object creation expression (§14.5.10.1) that passes the variable as an output parameter.
  - For a local variable, a local variable declaration (§15.5) 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. [*Note:* This ensures that undefined values never occur. *end note*] The occurrence of a variable in an expression is considered to obtain the value of the variable, except when
  - 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. [*Note:* This ensures that the function member being invoked can consider the reference parameter initially assigned. *end note*]
- All output parameters of a function member 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). [*Note:* This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable. *end note*]
- The `this` variable of a *struct-type* instance constructor must be definitely assigned at each location where that instance constructor returns.

12.3.1 Initially assigned variables

The following categories of variables are classified as initially assigned:

- Static variables.
- Instance variables of class instances.

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

12.3.2 Initially unassigned variables

The following categories of variables are classified as initially unassigned:

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

12.3.3 Precise rules for determining definite assignment

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 (§15.1) 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 the 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.

12.3.3.1 General rules for statements

- *v* is not definitely assigned at the beginning of a function member body.
- *v* is definitely assigned at the beginning of any unreachable statement.
- 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 (§15.1).

- The definite assignment state of *v* at the end point of a block, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using`, or `switch` statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the end point of that statement. If *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the end point of the statement. Otherwise, *v* is not definitely assigned at the end point of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (§15.1).

12.3.3.2 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.

12.3.3.3 Expression statements

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

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* 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*.

12.3.3.4 Declaration statements

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

12.3.3.5 If statements

For an `if` statement *stmt* of the form:

`if (`*expr*`)` *then-stmt* `else` *else-stmt*

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

12.3.3.6 Switch statements

In a `switch` statement *stmt* with 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*.

12.3.3.7 While statements

For a `while` statement *stmt* of the form:

`while (`*expr*`)` *while-body*

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

12.3.3.8 Do statements

For a `do` statement *stmt* of the form:

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

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

12.3.3.9 For statements

Definite assignment checking for a `for` statement of the form:

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

is done as if the statement were written:

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

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

12.3.3.10 Break, continue, and goto statements

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

12.3.3.11 Throw statements

For a statement *stmt* of the form

```
throw expr ;
```

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

12.3.3.12 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.

12.3.3.13 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*).

12.3.3.14 Try-finally statements

For a `try` statement *stmt* of the form:

```
try try-block finally finally-block
```

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

12.3.3.15 Try-catch-finally statements

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

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

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

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

[*Example:* The following example demonstrates how the different blocks of a `try` statement (§15.10) affect definite assignment.

```
class A
{
  static void F() {
     int i, j;
     try {
        goto LABEL:
        // neither i nor j definitely assigned
        i = 1;
        // i definitely assigned
     }
     catch {
        // neither i nor j definitely assigned
        i = 3;
        // i definitely assigned
     }
     finally {
        // neither i nor j definitely assigned
        j = 5;
        // j definitely assigned
     }
     // i and j definitely assigned
    LABEL:
     // j definitely assigned

  }
}
```

end example]

12.3.3.16 Foreach statements

For a `foreach` statement *stmt* of the form:

`foreach (`*type identifier* `in` *expr*`)` *embedded-statement*

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

12.3.3.17 Using statements

For a `using` statement *stmt* of the form:

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

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

12.3.3.18 Lock statements

For a `lock` statement *stmt* of the form:

`lock (`*expr*`)` *embedded-statement*

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

12.3.3.19 General rules for simple expressions

The following rule applies to these kinds of expressions: literals (§14.5.1), simple names (§14.5.2), member access expressions (§14.5.4), non-indexed base access expressions (§14.5.8), and `typeof` expressions (§14.5.11).

- 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.

12.3.3.20 General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions (§14.5.3), element access expressions (§14.5.6), base access expressions with indexing (§14.5.8), increment and decrement expressions (§14.5.9, §14.6.5), cast expressions (§14.6.6), unary `+`, `-`, `~`, `*` expressions, binary `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions (§14.7, §14.8, §14.9, §14.10), compound assignment expressions (§14.13.2), `checked` and `unchecked` expressions (§14.5.12), array and delegate creation expressions (§14.5.10).

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 $expr_1$, $expr_2$, ..., $expr_n$, evaluated in that order:

- The definite assignment state of *v* at the beginning of $expr_1$ is the same as the definite assignment state at the beginning of *expr*.
- The definite assignment state of *v* at the beginning of $expr_i$ (*i* greater than one) is the same as the definite assignment state at the end of $expr_{i-1}$.
- The definite assignment state of *v* at the end of *expr* is the same as the definite assignment state at the end of $expr_n$.

12.3.3.21 Invocation expressions and object creation expressions

For an invocation expression *expr* of the form:

primary-expression `(`arg_1, arg_2, …, arg_n`)`

or an object creation expression of the form:

`new` *type* $(arg_1, arg_2, \ldots, arg_n)$

- For an invocation expression, the definite assignment state of *v* before *primary-expression* is the same as the state of *v* before *expr*.
- For an invocation expression, the definite assignment state of *v* before arg_1 is the same as the state of *v* after *primary-expression*.
- For an object creation expression, the definite assignment state of *v* before arg_1 is the same as the state of *v* before *expr*.
- For each argument arg_i, the definite assignment state of *v* after arg_i is determined by the normal expression rules, ignoring any `ref` or `out` modifiers.
- For each argument arg_i for any *i* greater than one, the definite assignment state of *v* before arg_i is the same as the state of *v* after arg_{i-1}.
- If the variable *v* is passed as an `out` argument (i.e., an argument of the form "`out` *v*") in any of the arguments, then the state of *v* after *expr* is definitely assigned. Otherwise; the state of *v* after *expr* is the same as the state of *v* after arg_n.

12.3.3.22 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*.

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

[*Example:* In the 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's` being unassigned. *end example*]

12.3.3.24 || expressions

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

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

[*Example:* In the example

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

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

12.3.3.25 ! 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”.

12.3.3.26 ?: expressions

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

- The definite assignment state of *v* before *expr-cond* is the same as the state of *v* before *expr*.
- The definite assignment state of *v* before *expr-true* is definitely assigned if and only if the state of *v* after *expr-cond* is definitely assigned or “definitely assigned after true expression”.
- The definite assignment state of *v* before *expr-false* is definitely assigned if and only if the state of *v* after *expr-cond* is definitely assigned or “definitely assigned after false expression”.

12.4 Variable references

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

variable-reference:
  expression

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

12.5 Atomicity of variable references

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

13. Conversions

A ***conversion*** enables an expression of one type to be treated as another type. Conversions can be ***implicit*** or ***explicit***, and this determines whether an explicit cast is required. [*Example:* For instance, the conversion from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`. The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.

```
int a = 123;
long b = a;        // implicit conversion from int to long
int c = (int) b;  // explicit conversion from long to int
```

end example] Some conversions are defined by the language. Programs may also define their own conversions (§13.4).

13.1 Implicit conversions

The following conversions are classified as implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit enumeration conversions.
- Implicit reference conversions
- Boxing conversions
- Implicit constant expression conversions
- User-defined implicit conversions

Implicit conversions can occur in a variety of situations, including function member invocations (§14.4.3), cast expressions (§14.6.6), and assignments (§14.13).

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

13.1.1 Identity conversion

An identity conversion converts from any type to the same type. This conversion exists only such that an entity that already has a required type can be said to be convertible to that type.

13.1.2 Implicit numeric conversions

The implicit numeric conversions are:

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

- From `ulong` to `float`, `double`, or `decimal`.
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `float` to `double`.

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

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

13.1.3 Implicit enumeration conversions

An implicit enumeration conversion permits the *decimal-integer-literal* `0` to be converted to any *enum-type*.

13.1.4 Implicit reference conversions

The implicit reference conversions are:

- From any *reference-type* to `object`.
- From any *class-type* `S` to any *class-type* `T`, provided `S` is derived from `T`.
- From any *class-type* `S` to any *interface-type* `T`, provided `S` implements `T`.
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is derived from `T`.
- From an *array-type* `S` with an element type S_E to an *array-type* `T` with an element type T_E, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - Both S_E and T_E are *reference-types*.
  - An implicit reference conversion exists from S_E to T_E.
- From any *array-type* to `System.Array`.
- From any *delegate-type* to `System.Delegate`.
- From any *array-type* or *delegate-type* to `System.ICloneable`.
- From the null type to any *reference-type*.

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

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

13.1.5 Boxing conversions

A boxing conversion permits any *value-type* to be implicitly converted to the type `object` or to any *interface-type* implemented by the *value-type*. Boxing a value of a *value-type* consists of allocating an object instance and copying the *value-type* value into that instance.

Boxing conversions are described further in §11.3.1.

13.1.6 Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A *constant-expression* (§14.15) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the *constant-expression* is within the range of the destination type.

- A *constant-expression* of type `long` can be converted to type `ulong`, provided the value of the *constant-expression* is not negative.

13.1.7 User-defined implicit conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined conversions are described in §13.4.3.

13.2 Explicit conversions

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- User-defined explicit conversions.

Explicit conversions can occur in cast expressions (§14.6.6).

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

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

13.2.1 Explicit numeric conversions

The explicit numeric conversions are the conversions from a *numeric-type* to another *numeric-type* for which an implicit numeric conversion (§13.1.2) does not already exist:

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

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any *numeric-type* to any other *numeric-type* using a cast expression (§14.6.6).

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

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context (§14.5.12) in which the conversion takes place:
  - In a `checked` context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a `System.OverflowException` if the value of the source operand is outside the range of the destination type.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
    - If the source type is larger than the destination type, then the source value is truncated by discarding its “extra” most significant bits. The result is then treated as a value of the destination type.
    - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
    - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type
- For a conversion from `decimal` to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- For a conversion from `float`or `double` to an integral type, the processing depends on the overflow-checking context (§14.5.12) in which the conversion takes place:
  - In a `checked` context, the conversion proceeds as follows:
    - If the value of the source operand is within the range of the destination type, then it is rounded towards zero to the nearest integral value of the destination type, and this integral value is the result of the conversion.
    - Otherwise, a `System.OverflowException` is thrown.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
    - If the value of the source operand is within the range of the destination type, then it is rounded towards zero to the nearest integral value of the destination type, and this integral value is the result of the conversion.
    - Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes positive zero or negative zero. If the `double` value is too large to represent as a `float`, the result becomes positive infinity or negative infinity. If the `double` value is NaN, the result is also NaN.
- For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28^{th} decimal place if required (§11.1.6). If the source value is too small to represent as a `decimal`, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a `decimal`, a `System.OverflowException` is thrown.
- For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value. While this conversion may lose precision, it never causes an exception to be thrown.

13.2.2 Explicit enumeration conversions

The explicit enumeration conversions are:

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

An explicit enumeration conversion between two types is processed by treating any participating *enum-type* as the underlying type of that *enum-type*, and then performing an implicit or explicit numeric conversion between the resulting types. For example, given an *enum-type* `E` with and underlying type of `int`, a conversion from `E` to `byte` is processed as an explicit numeric conversion (§13.2.1) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion (§13.1.2) from `byte` to `int`.

13.2.3 Explicit reference conversions

The explicit reference conversions are:

- From `object` to any *reference-type*.
- From any *class-type* `S` to any *class-type* `T`, provided `S` is a base class of `T`.
- From any *class-type* `S` to any *interface-type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface-type* `S` to any *class-type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is not derived from `T`.
- From an *array-type* `S` with an element type S_E to an *array-type* `T` with an element type T_E, provided all of the following are true:
    - `S` and `T` differ only in element type. (In other words, `S` and `T` have the same number of dimensions.)
    - Both S_E and T_E are *reference-types*.
    - An explicit reference conversion exists from S_E to T_E.
- From `System.Array` and the interfaces it implements, to any *array-type*.
- From `System.Delegate` and the interfaces it implements, to any *delegate-type*.

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

For an explicit reference conversion to succeed at run-time, the value of the source operand must be `null`, or the *actual* type of the object referenced by the source operand must be a type that can be converted to the destination type by an implicit reference conversion (§13.1.4). If an explicit reference conversion fails, a `System.InvalidCastException` is thrown.

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

13.2.4 Unboxing conversions

An unboxing conversion permits an explicit conversion from type `object` to any *value-type* or from any *interface-type* to any *value-type* that implements the *interface-type*. An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance.

Unboxing conversions are described further in §11.3.2.

13.2.5 User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined conversions are described in §13.4.4.

13.3 Standard conversions

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

13.3.1 Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (§13.1.1)
- Implicit numeric conversions (§13.1.2)
- Implicit reference conversions (§13.1.4)
- Boxing conversions (§13.1.5)
- Implicit constant expression conversions (§13.1.6)

The standard implicit conversions specifically exclude user-defined implicit conversions.

13.3.2 Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. [*Note:* In other words, if a standard implicit conversion exists from a type `A` to a type `B`, then a standard explicit conversion exists from type `A` to type `B` and from type `B` to type `A`. *end note*]

13.4 User-defined conversions

C# allows the pre-defined implicit and explicit conversions to be augmented by ***user-defined conversions***. User-defined conversions are introduced by declaring conversion operators (§17.9.3) in class and struct types.

13.4.1 Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S` and `T` are different types.
- Either `S` or `T` is the class or struct type in which the operator declaration takes place.
- Neither `S` nor `T` is `object` or an *interface-type*.
- `T` is not a base class of `S`, and `S` is not a base class of `T`.

The restrictions that apply to user-defined conversions are discussed further in §17.9.3.

13.4.2 Evaluation of user-defined conversions

A user-defined conversion converts a value from its type, called the ***source type***, to another type, called the ***target type***. Evaluation of a user-defined conversion centers on finding the ***most specific*** user-defined conversion operator for the particular source and target types. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes and the target type and its base classes (with the implicit assumptions that only classes and structs can declare user-defined operators, and that non-class types have no base classes).
- From that set of types, determining which user-defined conversion operators are applicable. For a conversion operator to be applicable, it must be possible to perform a standard conversion (§13.3) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most specific. In general terms, the most specific operator is the operator whose operand type is “closest” to the source type and whose result type is “closest” to the target type. The exact rules for establishing the most specific user-defined conversion operator are defined in the following sections.

Once a most specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source type to the operand type of the user-defined conversion operator.
- Next, invoking the user-defined conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following sections. The definitions make use of the following terms:

- If a standard implicit conversion (§13.3.1) exists from a type `A` to a type `B`, and if neither `A` nor `B` are *interface-type*s, then `A` is said to be ***encompassed by*** `B`, and `B` is said to ***encompass*** `A`.
- The ***most encompassing type*** in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most encompassing type. In more intuitive terms, the most encompassing type is the “largest” type in the set—the one type to which each of the other types can be implicitly converted.
- The ***most encompassed type*** in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most encompassed type. In more intuitive terms, the most encompassed type is the “smallest” type in the set—the one type that can be implicitly converted to each of the other types.

13.4.3 User-defined implicit conversions

A user-defined implicit conversion from type `S` to type `T` is processed as follows:

- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S` (if `S` is a class or struct), the base classes of `S` (if `S` is a class), `T` (if `T` is a class or struct), and the base classes of `T` (if `T` is a class).
- Find the set of applicable user-defined conversion operators, U. This set consists of the user-defined implicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing `S` to a type encompassed by T. If `U` is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, S_X, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then S_X is `S`.

  - Otherwise, S_X is the most encompassed type in the combined set of source types of the operators in U. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, T_X, of the operators in U:
  - If any of the operators in U convert to T, then T_X is T.
  - Otherwise, T_X is the most encompassing type in the combined set of target types of the operators in U. If no most encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
- If U contains exactly one user-defined conversion operator that converts from S_X to T_X, then this is the most specific conversion operator. If no such operator exists, or if more than one such operator exists, then the conversion is ambiguous and a compile-time error occurs. Otherwise, the user-defined conversion is applied:
  - If S is not S_X, then a standard implicit conversion from S to S_X is performed.
  - The most specific user-defined conversion operator is invoked to convert from S_X to T_X.
  - If T_X is not T, then a standard implicit conversion from T_X to T is performed.

13.4.4 User-defined explicit conversions

A user-defined explicit conversion from type S to type T is processed as follows:

- Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S (if S is a class or struct), the base classes of S (if S is a class), T (if T is a class or struct), and the base classes of T (if T is a class).
- Find the set of applicable user-defined conversion operators, U. This set consists of the user-defined implicit or explicit conversion operators declared by the classes or structs in D that convert from a type encompassing or encompassed by S to a type encompassing or encompassed by T. If U is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, S_X, of the operators in U:
  - If any of the operators in U convert from S, then S_X is S.
  - Otherwise, if any of the operators in U convert from types that encompass S, then S_X is the most encompassed type in the combined set of source types of those operators. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, S_X is the most encompassing type in the combined set of source types of the operators in U. If no most encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, T_X, of the operators in U:
  - If any of the operators in U convert to T, then T_X is T.
  - Otherwise, if any of the operators in U convert to types that are encompassed by T, then T_X is the most encompassing type in the combined set of source types of those operators. If no most encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, T_X is the most encompassed type in the combined set of target types of the operators in U. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- If U contains exactly one user-defined conversion operator that converts from S_X to T_X, then this is the most specific conversion operator. If no such operator exists, or if more than one such operator exists, then the conversion is ambiguous and a compile-time error occurs. Otherwise, the user-defined conversion is applied:

- If `S` is not S_X, then a standard explicit conversion from `S` to S_X is performed.
- The most specific user-defined conversion operator is invoked to convert from S_X to T_X.
- If T_X is not `T`, then a standard explicit conversion from T_X to `T` is performed.

14. 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.

14.1 Expression classifications

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left-hand side of a *member-access* (§14.5.4). 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* (§14.5.4), or as an operand for the `as` operator (§14.9.10), the `is` operator (§14.9.9), or the `typeof` operator (§14.5.11). 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 (§14.3). A method group may have an associated instance expression. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (§14.5.7). A method group is only permitted in an *invocation-expression* (§14.5.5) or a *delegate-creation-expression* (§14.5.10.3). In any other context, an expression classified as a method group causes a compile-time error.
- 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` (§14.5.7).
- 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 (§14.13.3). 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` (§14.5.7), 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* (§15.6).

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 (§14.13.1). Otherwise, the *get-accessor* is invoked to obtain the current value (§14.1.1).

14.1.1 Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a ***value***. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned (§12.3) before its value can be obtained, or otherwise a compile-time error occurs.
- The value of a property access expression is obtained by invoking the *get-accessor* of the property. If the property has no *get-accessor*, a compile-time error occurs. Otherwise, a function member invocation (§14.4.3) is performed, and the result of the invocation becomes the value of the property access expression.
- The value of an indexer access expression is obtained by invoking the *get-accessor* of the indexer. If the indexer has no *get-accessor*, a compile-time error occurs. Otherwise, a function member invocation (§14.4.3) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

14.2 Operators

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

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as `–x`) or postfix notation (such as `x++`).
- Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).
- Ternary operator. Only one ternary operator, `?:`, exists; it takes three operands and uses infix notation (`c ? x : y`).

The order of evaluation of operators in an expression is determined by the ***precedence*** and ***associativity*** of the operators (§14.2.1).

The order in which operands in an expression are evaluated, is left to right. [*Example:* For example, in `F(i) + G(i++) * H(i)`, method `F` is called using the old value of `i`, then method `G` is called with the old value of `i`, and, finally, method `H` is called with the new value of `i`. This is separate from and unrelated to operator precedence. *end example*] Certain operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (§14.2.2).

14.2.1 Operator precedence and associativity

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. [*Note:* For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. *end note*] The precedence of an operator is established by the definition of its associated grammar production. [*Note:* For example, an *additive-expression* consists of a sequence of *multiplicative-expression*s separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*`, `/`, and `%` operators. *end note*]

The following table summarizes all operators in order of precedence from highest to lowest:

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

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

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

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

14.2.2 Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including `operator` declarations in classes and structs (§17.9). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

The ***overloadable unary operators*** are:

```
+   -   !   ~   ++   --   true   false
```

[*Note*: Although `true` and `false` are not used explicitly in expressions, they are considered operators because they are invoked in several expression contexts: boolean expressions (§14.16) and expressions involving the conditional (§14.12), and conditional logical operators (§14.11). *end note*]

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, 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 §14.13. 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 (§13.4).

Element access, such as a[x], is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (§17.8).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix ++ and -- operators. In the third entry, *op* denotes any overloadable binary operator. [*Note:* For an example of overloading the ++ and -- operators see §17.9.1. *end note*]

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. [*Note:* Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator. *end note*]

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. [*Example:* For example, the / operator is always a binary operator, always has the precedence level specified in §14.2.1, and is always left-associative. *end example*]

[*Note:*While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of operator == should compare the two operands for equality and return an appropriate bool result. *end note*]

The descriptions of individual operators in §14.5 through §14.13 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.

14.2.3 Unary operator overload resolution

An operation of the form *op* x or x *op*, where *op* is an overloadable unary operator, and x is an expression of type X, is processed as follows:

- The set of candidate user-defined operators provided by X for the operation operator *op*(x) is determined using the rules of §14.2.5.
- 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 become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§14.5 and §14.6).

- The overload resolution rules of §14.4.2 are applied to the set of candidate operators to select the best operator with respect to the argument list `(x)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a compile-time error occurs.

14.2.4 Binary operator overload resolution

An operation of the form `x` *op* `y`, where *op* is an overloadable binary operator, `x` is an expression of type `X`, and `y` is an expression of type `Y`, is processed as follows:

- The set of candidate user-defined operators provided by `X` and `Y` for the operation `operator` *op*`(x, y)` is determined. The set consists of the union of the candidate operators provided by `X` and the candidate operators provided by `Y`, each determined using the rules of §14.2.5. If `X` and `Y` are the same type, or if `X` and `Y` are derived from a common base type, then shared candidate operators only occur in the combined set once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator` *op* implementations become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (§14.7 through §14.13).
- The overload resolution rules of §14.4.2 are applied to the set of candidate operators to select the best operator with respect to the argument list `(x, y)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a compile-time error occurs.

14.2.5 Candidate user-defined operators

Given a type `T` and an operation `operator` *op*`(A)`, where *op* is an overloadable operator and `A` is an argument list, the set of candidate user-defined operators provided by `T` for `operator` *op*`(A)` is determined as follows:

- For all `operator` *op* declarations in `T`, if at least one operator is applicable (§14.4.2.1) with respect to the argument list `A`, then the set of candidate operators consists of all applicable `operator` *op* declarations in `T`.
- Otherwise, if `T` is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by `T` is the set of candidate operators provided by the direct base class of `T`.

14.2.6 Numeric promotions

This clause is informative.

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary * operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules (§14.4.2) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. [*Example:* For example, for the operation b * s, where b is a byte and s is a short, overload resolution selects operator *(int, int) as the best operator. Thus, the effect is that b and s are converted to int, and the type of the result is int. Likewise, for the operation i * d, where i is an int and d is a double, overload resolution selects operator *(double, double) as the best operator. *end example*]

End of informative text.

14.2.6.1 Unary numeric promotions

This clause is informative.

Unary numeric promotion occurs for the operands of the predefined +, −, and ~ unary operators. Unary numeric promotion simply consists of converting operands of type sbyte, byte, short, ushort, or char to type int. Additionally, for the unary − operator, unary numeric promotion converts operands of type uint to type long.

End of informative text.

14.2.6.2 Binary numeric promotions

This clause is informative.

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 compile-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 compile-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:* 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. *end note*]

[*Note:* Also note that it is not possible for an operand to be of type ulong when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of ulong as well as the signed integral types. *end note*]

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

[*Example:* In the example

```
decimal AddPercent(decimal x, double percent) {
   return x * (1.0 + percent / 100.0);
}
```

a compile-time error occurs because a decimal cannot be multiplied by a double. The error is resolved by explicitly converting the second operand to decimal, as follows:

```
    decimal AddPercent(decimal x, double percent) {
        return x * (decimal)(1.0 + percent / 100.0);
    }
```

end example]

End of informative text.

14.3 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 may occur as part of evaluating a *simple-name* (§14.5.2) or a *member-access* (§14.5.4) in an expression.

A member lookup of a name `N` in a type `T` is processed as follows:

- First, the set of all accessible (§10.5) members named `N` declared in `T` and the base types (§14.3.1) of `T` is constructed. Declarations that include an `override` modifier are excluded from the set. If no members named `N` exist and are accessible, then the lookup produces no match, and the following steps are not evaluated.
- 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, type, or enumeration member, then all members 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, and all methods with the same signature as `M` declared in a base type of `S` are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined:
  - If the set consists of a single non-method member, 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 compile-time error occurs (this situation can only occur for a member lookup in an interface that has multiple direct base interfaces).

For member lookups in types other than 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 §20.2.5.

14.3.1 Base types

For purposes of member lookup, a type `T` is considered to have the following base types:

- If `T` is `object`, then `T` has no base type.
- If `T` is a *value-type*, the base type of `T` is the class type `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`.

14.4 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 static constructors and destructors (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 (§14.4.1) of a function member invocation provides actual values or variable references for the parameters of the function member.

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 §14.4.2.

Once a particular function member has been identified at compile-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in §14.4.3.

[*Note:* The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the table, `e`, `x`, `y`, and `value` indicate expressions classified as variables or values, `T` indicates an expression classified as a type, `F` is the simple name of a method, and `P` is the simple name of a property.

Construct	**Example**	**Description**
Method invocation	`F(x, y)`	Overload resolution is applied to select the best method `F` in the containing class or struct. The method is invoked with the argument list `(x, y)`. If the method is not `static`, the instance expression is `this`.
	`T.F(x, y)`	Overload resolution is applied to select the best method `F` in the class or struct T. A compile-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 compile-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.

Construct	Example	Description
	`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 compile-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 compile-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 compile-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 compile-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 compile-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 compile-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 compile-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 compile-time error occurs if the indexer is read-only.
Operator invocation	`-x`	Overload resolution is applied to select the best unary operator in the class or struct given by the type of `x`. The selected operator is invoked with the argument list `(x)`.
	`x + y`	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of x and y. The selected operator is invoked with the argument list `(x, y)`.
Instance constructor invocation	`new T(x, y)`	Overload resolution is applied to select the best instance constructor in the class or struct T. The instance constructor is invoked with the argument list `(x, y)`.

end note]

14.4.1 Argument lists

Every function member 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, and delegates, the arguments are specified as an *argument-list*, as described below.
- 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 indexers, the argument list consists of the expressions specified between the square brackets in the indexer access. When invoking the `set` accessor, the argument list additionally includes the expression specified as the right operand of the assignment 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 (§17.6), events (§17.7), indexers (§17.8), and user-defined operators (§17.9) are always passed as value parameters (§17.5.1.1). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, or delegate invocation are specified as an *argument-list*:

argument-list:
argument
argument-list `,` *argument*

argument:
expression
`ref` *variable-reference*
`out` *variable-reference*

An *argument-list* consists of one or more *argument*s, separated by commas. Each argument can take one of the following forms:

- An *expression*, indicating that the argument is passed as a value parameter (§17.5.1.1).
- The keyword `ref` followed by a *variable-reference* (§12.3.3), indicating that the argument is passed as a reference parameter (§17.5.1.2). A variable must be definitely assigned (§12.3) before it can be passed as a reference parameter. A volatile field (§17.4.3) cannot be passed as a reference parameter.
- The keyword `out` followed by a *variable-reference* (§12.3.3), indicating that the argument is passed as an output parameter (§17.5.1.3). A variable is considered definitely assigned (§12.3) following a function member invocation in which the variable is passed as an output parameter. A volatile field (§17.4.3) cannot be passed as an output parameter.

During the run-time processing of a function member invocation (§14.4.3), 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 (§13.1) 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 (§17.5.1.4). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (§14.4.2.1):

- 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 of a type that is implicitly convertible (§13.1) 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 arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible (§13.1) 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. [*Example:* Thus, the example

```
class Test
{
   static void F(int x, int y, int z) {
      System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
   }
   static void Main() {
      int i = 0;
      F(i++, i++, i++);
   }
}
```

produces the output

```
x = 0, y = 1, z = 2
```

end example]

The array covariance rules (§19.5) 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. [*Example:* 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`. *end example*]

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (§14.5.10.2) was inserted around the expanded parameters. [*Example:* 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});
```

end example] In particular, note that an empty array is created when there are zero arguments given for the parameter array.

14.4.2 Overload resolution

Overload resolution is a compile-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* (§14.5.5).
- Invocation of an instance constructor named in an *object-creation-expression* (§14.5.10.1).
- Invocation of an indexer accessor through an *element-access* (§14.5.6).
- Invocation of a predefined or user-defined operator referenced in an expression (§14.2.3 and §14.2.4).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way. However, once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- First, the set of candidate function members is reduced to those function members that are applicable with respect to the given argument list (§14.4.2.1). If this reduced set is empty, a compile-time error occurs.
- Then, 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 §14.4.2.2. 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 compile-time error occurs.

The following sections define the exact meanings of the terms ***applicable function member*** and ***better function member***.

14.4.2.1 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:

- The number of arguments in `A` is identical to the number of parameters in the function member declaration.
- 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 (§13.1) 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. [*Note:* After all, a `ref` or `out` parameter is an alias for the argument passed. *end note*]

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.
- If the class, struct, or interface in which the function member is declared already contains another applicable function member with the same signature as the expanded form, the expanded form is 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 (§13.1) 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.

14.4.2.2 Better function member

Given an argument list `A` with a set of argument types A_1, A_2, …, A_N and two applicable function members M_P and M_Q with parameter types P_1, P_2, …, P_N and Q_1, Q_2, …, Q_N, M_P is defined to be a ***better function member*** than M_Q if

- for each argument, the implicit conversion from A_X to P_X is not worse than the implicit conversion from A_X to Q_X, and
- for at least one argument, the conversion from A_X to P_X is better than the conversion from A_X to Q_X.

When performing this evaluation, if M_P or M_Q is applicable in its expanded form, then P_X or Q_X refers to a parameter in the expanded form of the parameter list.

14.4.2.3 Better conversion

Given an implicit conversion C_1 that converts from a type `S` to a type T_1, and an implicit conversion C_2 that converts from a type `S` to a type T_2, the ***better conversion*** of the two conversions is determined as follows:

- If T_1 and T_2 are the same type, neither conversion is better.
- If `S` is T_1, C_1 is the better conversion.
- If `S` is T_2, C_2 is the better conversion.
- If an implicit conversion from T_1 to T_2 exists, and no implicit conversion from T_2 to T_1 exists, C_1 is the better conversion.
- If an implicit conversion from T_2 to T_1 exists, and no implicit conversion from T_1 to T_2 exists, C_2 is the better conversion.
- If T_1 is `sbyte` and T_2 is `byte`, `ushort`, `uint`, or `ulong`, C_1 is the better conversion.
- If T_2 is `sbyte` and T_1 is `byte`, `ushort`, `uint`, or `ulong`, C_2 is the better conversion.
- If T_1 is `short` and T_2 is `ushort`, `uint`, or `ulong`, C_1 is the better conversion.
- If T_2 is `short` and T_1 is `ushort`, `uint`, or `ulong`, C_2 is the better conversion.
- If T_1 is `int` and T_2 is `uint`, or `ulong`, C_1 is the better conversion.
- If T_2 is `int` and T_1 is `uint`, or `ulong`, C_2 is the better conversion.
- If T_1 is `long` and T_2 is `ulong`, C_1 is the better conversion.
- If T_2 is `long` and T_1 is `ulong`, C_2 is the better conversion.

- Otherwise, neither conversion is better.

If an implicit conversion C_1 is defined by these rules to be a better conversion than an implicit conversion C_2, then it is also the case that C_2 is a ***worse conversion*** than C_1.

14.4.3 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 compile-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are static methods, instance constructors, 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` (§14.5.7).

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 §14.4.1.
  - `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 §14.4.1.
  - `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 §14.4.1.
  - If the type of `E` is a *value-type*, a boxing conversion (§11.3.1) is performed to convert `E` to type `object`, and `E` is considered to be of type `object` in the following steps. [*Note:* In this case, `M` could only be a member of `System.Object`. *end note*]
  - 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 compile-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 (§20.4.2) 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 (§17.5.3) 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.

o The function member implementation determined in the step above is invoked. The object referenced by `E` becomes the object referenced by `this`.

14.4.3.1 Invocations on boxed instances

A function member implemented in a *value-type* can be invoked through a boxed instance of that *value-type* in the following situations:

- When the function member is an `override` of a method inherited from type `object` and is invoked through an instance expression of type `object`.
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface-type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value-type*, and this variable becomes the variable referenced by `this` within the function member invocation. [*Note:* In particular, this means that when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance. *end note*]

14.5 Primary expressions

Primary expressions include the simplest forms of expressions.

primary-expression:
  array-creation-expression
  primary-no-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
  typeof-expression
  sizeof-expression
  checked-expression
  unchecked-expression

Primary expressions are divided between *array-creation-expression*s and *primary-no-array-creation-expression*s. 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];
```

14.5.1 Literals

A *primary-expression* that consists of a *literal* (§9.4.4) is classified as a value.

14.5.2 Simple names

A *simple-name* consists of a single identifier.

simple-name:
    identifier

A *simple-name* is evaluated and classified as follows:

- If the *simple-name* appears within a *block* and if the *block*'s (or an enclosing block's) local variable declaration space (§10.3) contains a local variable or parameter with the given name, then the *simple-name* refers to that local variable or parameter and is classified as a variable.
- Otherwise, for each type T, starting with the immediately enclosing class, struct, or enumeration declaration and continuing with each enclosing outer class or struct declaration (if any), if a member lookup of the *simple-name* in `T` produces a match:
  - If `T` is 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 `T` is 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 (§14.5.4) of the form `this.E`, where `E` is the *simple-name*.
  - Otherwise, the result is the same as a member access (§14.5.4) of the form `T.E`, where `E` is the *simple-name*. In this case, it is a compile-time error for the *simple-name* to refer to an instance member.
- Otherwise, 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 the namespace contains a namespace member with the given name, then the *simple-name* refers to that member and, depending on the member, is classified as a namespace or a type.
  - Otherwise, if the namespace has a corresponding namespace declaration enclosing the location where the *simple-name* occurs, then:
    - If the namespace declaration contains a *using-alias-directive* that associates the given name with an imported namespace or type, then the *simple-name* refers to that namespace or type.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type with the given name, then the *simple-name* refers to that type.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type with the given name, then the *simple-name* is ambiguous and a compile-time error occurs.
- Otherwise, the name given by the *simple-name* is undefined and a compile-time error occurs.

14.5.2.1 Invariant meaning in blocks

For each occurrence of a given identifier as a *simple-name* in an expression, every other occurrence of the same identifier as a *simple-name* in an expression within the immediately enclosing *block* (§15.2) or *switch-block* (§15.7.2) must refer to the same entity. This rule ensures that the meaning of a name in the context of an expression is always the same within a block.

The example

```
class Test
{
   double x;
   void F(bool b) {
      x = 1.0;
      if (b) {
         int 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 = 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 (§14.5.4). [*Example:* 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. *end example*]

14.5.3 Parenthesized expressions

A *parenthesized-expression* consists of an *expression* enclosed in parentheses.

parenthesized-expression:
    `(` *expression* `)`

A *parenthesized-expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a namespace, type, or method group, a compile-time error occurs. Otherwise, the result of the *parenthesized-expression* is the result of the evaluation of the contained *expression*.

14.5.4 Member access

A *member-access* consists of a *primary-expression* or a *predefined-type*, followed by a "`.`" token, followed by an *identifier*.

member-access:
  primary-expression `.` *identifier*
  predefined-type `.` *identifier*

predefined-type: one of

```
bool     byte     char     decimal  double   float    int      long
object   sbyte    short    string   uint     ulong    ushort
```

A *member-access* of the form `E.I`, where `E` is a *primary-expression* or a *predefined-type* and `I` is an *identifier*, is evaluated and classified as follows:

- If `E` is a namespace and `I` is the name of an accessible member of that namespace, then the result is that member and, depending on the member, is classified as a namespace or a type.
- If `E` is a *predefined-type* or a *primary-expression* classified as a type, and a member lookup (§14.3) of `I` in `E` produces a match, then `E.I` is evaluated and classified as follows:
  - If `I` identifies a type, then the result is that type.
  - If `I` identifies one or more methods, then the result is a method group with no associated instance expression.
  - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
  - If `I` identifies a `static` field:
    - If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field `I` in E.
    - Otherwise, the result is a variable, namely the static field `I` in E.
  - If `I` identifies a `static` event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declaration*s (§17.7), then `E.I` is processed exactly as if `I` was 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 (§14.3) of `I` in `T` 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 (§14.1.1) 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 `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-declaration*s (§17.7), 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, `E.I` is an invalid member reference, and a compile-time error occurs.

14.5.4.1 Identical simple names and type names

In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a *simple-name* (§14.5.2) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as a *type-name* (§10.8), 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 of `E` where a compile-time error would otherwise have occurred. [*Example:* 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. *end example*]

14.5.5 Invocation expressions

An *invocation-expression* is used to invoke a method.

invocation-expression:
  primary-expression `(` *argument-list*$_{opt}$ `)`

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 (§14.5.5.1). If the *primary-expression* is a value of a *delegate-type*, the *invocation-expression* is a delegate invocation (§14.5.5.2). If the *primary-expression* is neither a method group nor a value of a *delegate-type*, a compile-time error occurs.

The optional *argument-list* (§14.4.1) 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 cannot be an operand of any operator, and is permitted only in the context of a *statement-expression* (§15.6).
- Otherwise, the result is a value of the type returned by the method or delegate.

14.5.5.1 Method invocations

For a method invocation, the *primary-expression* of the *invocation-expression* must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The compile-time processing of a method invocation of the form `M(A)`, where `M` is a method group and `A` is an optional *argument-list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. Starting with the set of methods associated with M, which were found by a previous member lookup (§14.3), the set is reduced to those methods that are applicable with respect to the argument list `A`. The set reduction consists of applying the following rules to each method `T.N` in the set, where `T` is the type in which the method `N` is declared:
  - If `N` is not applicable with respect to `A` (§14.4.2.1), then `N` is removed from the set.
  - If `N` is applicable with respect to `A` (§14.4.2.1), then all methods declared in a base type of `T` are removed from the set.
- If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time error occurs. If the candidate methods are not all declared in the same type, the method invocation is ambiguous, and a compile-time error occurs (this latter situation can only occur for an invocation of a method in an interface that has multiple direct base interfaces, as described in §20.2.5).
- The best method of the set of candidate methods is identified using the overload resolution rules of §14.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a compile-time error occurs.
- Given a best method, the invocation of 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 are true, a compile-time error occurs.

Once a method has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in §14.4.3.

[*Note:* The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. *end note*]

14.5.5.2 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 (§14.4.2.1) 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. A function member invocation (§14.4.3) is performed on the method referenced by the delegate. If the method is an instance method, the instance of the invocation becomes the instance referenced by the delegate.

14.5.6 Element access

An *element-access* consists of a *primary-no-array-creation-expression*, followed by a “`[`“ token, followed by an *expression-list*, followed by a “`]`” token. The *expression-list* consists of one or more *expression*s, separated by commas.

element-access:
  primary-no-array-creation-expression `[` *expression-list* `]`

expression-list:
  expression
  expression-list `,` *expression*

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 (§14.5.6.1). 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 (§14.5.6.2).

14.5.6.1 Array access

For an array access, the *primary-no-array-creation-expression* of the *element-access* must be a value of an *array-type*. The number of expressions in the *expression-list* must be the same as the rank of the *array-type*, and each expression must be of type `int`, `uint`, `long`, `ulong`, or of a type that can be implicitly converted to one or more of these types.

The result of 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 *expression-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 *expression-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 *expression-list* are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion (§13.1) 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 *expression-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.

14.5.6.2 Indexer access

For an indexer access, the *primary-no-array-creation-expression* of the *element-access* must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the *expression-list* of the *element-access*.

The compile-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 *expression-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 (§10.5).
- 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` (§14.4.2.1), then `I` is removed from the set.
  - If `I` is applicable with respect to `A` (§14.4.2.1), then all indexers declared in a base type of `S` are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a compile-time error occurs. If the candidate indexers are not all declared in the same type, the indexer access is ambiguous, and a compile-time error occurs (this latter situation can only occur for an indexer access on an instance of an interface that has multiple direct base interfaces).
- The best indexer of the set of candidate indexers is identified using the overload resolution rules of §14.4.2. If a single best indexer cannot be identified, the indexer access is ambiguous, and a compile-time error occurs.
- The index expressions of the *expression-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 (§14.13.1). In all other cases, the *get-accessor* is invoked to obtain the current value (§14.1.1).

14.5.7 This access

A *this-access* consists of the reserved word `this`.

> *this-access:*
>   `this`

A *this-access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When `this` is used in a *primary-expression* within an instance constructor of a class, it is classified as a value. The type of the value is the 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 class within which the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.
- When `this` is used in a *primary-expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
- When `this` is used in a *primary-expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the struct within which the usage occurs, and the variable represents the struct for which the method or accessor was invoked. The `this` variable of an instance method of a struct behaves exactly the same as a `ref` parameter of the struct type.

Use of `this` in a *primary-expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable-initializer* of a field declaration.

14.5.8 Base access

A *base-access* consists of the reserved word `base` followed by either a "`.`" token and an identifier or an *expression-list* enclosed in square brackets:

base-access:
  `base . ` *identifier*
  `base [ ` *expression-list* ` ]`

A *base-access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base-access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. When `base.I` occurs in a class or struct, `I` must denote a member of the base class of that class or struct. Likewise, when `base[E]` occurs in a class, an applicable indexer must exist in the base class.

At compile-time, *base-access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus, `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

When a *base-access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at run-time (§14.4.3) is changed. The function member that is invoked is determined by finding the most derived implementation (§17.5.3) of the function member with respect to `B` (instead of with respect to the run-time type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base-access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base-access* is abstract, a compile-time error occurs.

14.5.9 Postfix increment and decrement operators

post-increment-expression:
  primary-expression `++`

post-decrement-expression:
  primary-expression `--`

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a compile-time error occurs.

Unary operator overload resolution (§14.2.3) 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.

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 (§14.6.5). 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.

14.5.10 The `new` operator

The `new` operator is used to create new instances of types.

There are three forms of `new` expressions:

- Object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The `new` operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when `new` is used to create instances of value types.

14.5.10.1 Object creation expressions

An *object-creation-expression* is used to create a new instance of a *class-type* or a *value-type*.

object-creation-expression:
`new` *type* `(` *argument-list*$_{opt}$ `)`

The *type* of an *object-creation-expression* must be a *class-type* or a *value-type*. The *type* cannot be an `abstract` *class-type*.

The optional *argument-list* (§14.4.1) is permitted only if the *type* is a *class-type* or a *struct-type*.

The compile-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 §11.1.1.
- 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 §14.4.2. The set of candidate instance constructors consists of all accessible instance constructors declared in T. If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a compile-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 compile-time error occurs.

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 (§12.2).
  - The instance constructor is invoked according to the rules of function member invocation (§14.4.3). 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 (§14.4.3). 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`.

14.5.10.2 Array creation expressions

An *array-creation-expression* is used to create a new instance of an *array-type*.

array-creation-expression:
  `new` *non-array-type* `[` *expression-list* `]` *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
  `new` *array-type* *array-initializer*

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 of a type that can be implicitly converted 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 (§25.1), 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 form, the rank of the specified array type 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}}
```

Array initializers are described further in §19.6.

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 (§13.1) 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 (§12.2).
- 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. [*Example:* 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`. *end example*] 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.

14.5.10.3 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 (§14.1) or a value of 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 a value of a *delegate-type*, it identifies a delegate instance of which to create a copy.

The compile-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 set of methods identified by `E` must include exactly one method that is compatible (§22.1) with `D`, and this method becomes the one to which the newly created delegate refers. If no matching method exists, or if more than one matching method exists, a compile-time error occurs. If the selected method is an instance method, the instance expression associated with `E` determines the target object of the delegate.
  - As in a method invocation, the selected method must be compatible with the context of the method group: If the method is a static method, the method group must have resulted from a *simple-name* or a *member-access* through a type. If the method is an instance method, the method group must have resulted from a *simple-name* or a *member-access* through a variable or value. If the selected method does not match the context of the method group, a compile-time error occurs.
  - The result is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.
- Otherwise, if `E` is a value of a *delegate-type*:
  - `D` and `E` must be compatible (§22.1); otherwise, a compile-time error occurs.
  - The result is a value of type `D`, namely a newly created delegate that refers to the same invocation list as `E`.
- Otherwise, the delegate creation expression is invalid, and 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:
  - If the method selected at compile-time is a static method, the target object of the delegate is `null`. Otherwise, the selected method is an instance method, and the target object of the delegate is determined from the instance expression associated with `E`:
    - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
    - If the instance expression is of a *reference-type*, the value computed by the instance expression becomes the target object. If the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
    - If the instance expression is of a *value-type*, a boxing operation (§11.3.1) is performed to convert the value to an object, and this object becomes the target object.

  - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - The new delegate instance is initialized with a reference to the method that was determined at compile-time and a reference to the target object computed above.
- 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 references to the same invocation list as the delegate instance given by E.

The method and object to which a delegate refers are determined when the delegate is instantiated and then remain constant for the entire lifetime of the delegate. In other words, it is not possible to change the target method or object of a delegate once it has been created. [*Note:* Remember, when two delegates are combined or one is removed from another, a new delegate results; no existing delegate has its content changed. *end note*]

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

[*Example:* As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);

class A
{
   DoubleFunc f = new DoubleFunc(Square);

   static float Square(float x) {
      return x * x;
   }

   static double Square(double x) {
      return x * x;
   }
}
```

the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method not been present, a compile-time error would have occurred. *end example*]

14.5.11 The `typeof` operator

The `typeof` operator is used to obtain the `System.Type` object for a type.

typeof-expression:
   `typeof (` *type* `)`
   `typeof ( void )`

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. [*Note:* This means that for type `T`, `typeof(T) == typeof(T)` is always true. *end note*]

The second form of *typeof-expression* consists of a `typeof` keyword followed by a parenthesized `void` keyword. The result of an expression of this form is the `System.Type` object that represents the absence of a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type. [*Note:* This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of `System.Type`. *end note*]

[*Example:* The example

```
using System;
class Test
{
   static void Main() {
      Type[] t = {
         typeof(int),
         typeof(System.Int32),
         typeof(string),
         typeof(double[]),
         typeof(void)      };
      for (int i = 0; i < t.Length; i++) {
         Console.WriteLine(t[i].FullName);
      }
   }
}
```

produces the following output:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
```

Note that `int` and `System.Int32` are the same type. *end example*]

14.5.12 The checked and unchecked operators

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

checked-expression:
`checked` `(` *expression* `)`

unchecked-expression:
`unchecked` `(` *expression* `)`

The `checked` operator evaluates the contained expression in a checked context, and the `unchecked` operator evaluates the contained expression in an unchecked context. A *checked-expression* or *unchecked-expression* corresponds exactly to a *parenthesized-expression* (§14.5.3), 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 (§15.11).

The following operations are affected by the overflow checking context established by the `checked` and `unchecked` operators and statements:

- The predefined `++` and `--` unary operators (§14.5.9 and §14.6.5), when the operand is of an integral type.
- The predefined `-` unary operator (§14.6.2), when the operand is of an integral type.
- The predefined `+`, `-`, `*`, and `/` binary operators (§14.7), when both operands are of integral types.
- Explicit numeric conversions (§13.2.1) from one integral type to another 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 (§14.15), 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.

[*Note:* Developers may benefit if they exercise their code using checked mode (as well as unchecked mode). It also seems reasonable that, unless otherwise requested, the default overflow checking context is set to checked when debugging is enabled. *end note*]

[*Example:* 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`. *end example*]

[*Example:* 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. *end example*]

The `checked` and `unchecked` operators only affect the overflow checking context for those operations that are textually contained within the "`(`" and "`)`" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. [*Example:* In the 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. *end example*]

The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal notation. [*Example:* 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. *end example*]

[*Note:* The `checked` and `unchecked` operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow *even* within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an exception on overflow *even* within an explicitly `checked` construct. In addition, other operators are *never* affected by the mode of checking, whether default or explicit. As a service to programmers, it is recommended that the compiler issue a warning when there is an arithmetic expression within an explicitly `checked` or `unchecked` context (by operator or statement) that cannot possibly be affected by the specified mode of checking. Since such a warning is not required, the compiler has flexibility in determining the circumstances that merit the issuance of such warnings. *end note*]

14.6 Unary expressions

unary-expression:
  primary-expression
  `+` *unary-expression*
  `-` *unary-expression*
  `!` *unary-expression*
  `~` *unary-expression*
  `*` *unary-expression*
  `&` *unary-expression*
  pre-increment-expression
  pre-decrement-expression
  cast-expression

14.6.1 Unary plus operator

For an operation of the form `+x`, unary operator overload resolution (§14.2.3) 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.

14.6.2 Unary minus operator

For an operation of the form `–x`, unary operator overload resolution (§14.2.3) 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. In a `checked` context, if the value of `x` is the maximum negative `int` or `long`, a `System.OverflowException` is thrown. In an `unchecked` context, if the value of `x` is the maximum negative `int` or `long`, the result is that same value and the overflow is not reported.

  If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the result is `long`. An exception is the rule that permits the `int` value −2147483648 (-2^{31}) to be written as a decimal integer literal (§9.4.4.2).

  If the operand of the negation operator is of type `ulong`, a compile-time error occurs. An exception is the rule that permits the `long` value −9223372036854775808 (-2^{63}) to be written as a decimal integer literal (§9.4.4.2).

- 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`.

14.6.3 Logical negation operator

For an operation of the form `!x`, unary operator overload resolution (§14.2.3) 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`.

14.6.4 Bitwise complement operator

For an operation of the form `~x`, unary operator overload resolution (§14.2.3) 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)`.

14.6.5 Prefix increment and decrement operators

pre-increment-expression:
    `++` *unary-expression*

pre-decrement-expression:
    `--` *unary-expression*

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a compile-time error occurs.

Unary operator overload resolution (§14.2.3) 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.

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 (§14.5.9). 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.

14.6.6 Cast expressions

A *cast-expression* is used to explicitly convert an expression to a given type.

cast-expression:
`(` *type* `)` *unary-expression*

A *cast-expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary-expression*, performs an explicit conversion (§13.2) of the value of `E` to type `T`. If no explicit conversion exists from the type of `E` to `T`, a compile-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 *token*s (§9.4) enclosed in parentheses is considered the start of a *cast-expression* only if at least one of the following are true:

- The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
- The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing parentheses is the token "~", the token "`!`", the token "`(`", an *identifier* (§9.4.1), a *literal* (§9.4.4), or any *keyword* (§9.4.3) except `as` and `is`.

[*Note:* The above rule means that only if the construct is unambiguously a *cast-expression* is it considered a *cast-expression*. *end note*]

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.

[*Note:* From the disambiguation rule, it follows that, if `x` and `y` are identifiers, `(x)y`, `(x)(y)`, and `(x)(-y)` are *cast-expression*s, but `(x)-y` is not, even if `x` identifies a type. However, if `x` is a keyword that identifies a predefined type (such as `int`), then all four forms are *cast-expression*s (because such a keyword could not possibly be an expression by itself). *end note*]

14.7 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*

14.7.1 Multiplication operator

For an operation of the form `x * y`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of `x` and `y`.

- Integer multiplication:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

  In a `checked` context, if the product is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point multiplication:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

  The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+y	−y	+0	−0	+∞	−∞	NaN
+x	+z	−z	+0	−0	+∞	−∞	NaN
−x	−z	+z	−0	+0	−∞	+∞	NaN
+0	+0	−0	+0	−0	NaN	NaN	NaN
−0	−0	+0	−0	+0	NaN	NaN	NaN
+∞	+∞	−∞	NaN	NaN	+∞	−∞	NaN
−∞	−∞	+∞	NaN	NaN	−∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

  If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

  Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

14.7.2 Division operator

For an operation of the form `x / y`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of `x` and `y`.

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

The division rounds the result towards zero, and the absolute value of the result is the largest possible integer that is less than 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 maximum negative `int` or `long` value and the right operand is `–1`, an overflow occurs. In a `checked` context, this causes a `System.OverflowException` to be thrown. In an `unchecked` context, the overflow is not reported and the result is instead the value 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, before any rounding, is the smallest scale that will preserve a result equal to the exact result.

Decimal division is equivalent to using the division operator of type `System.Decimal`.

14.7.3 Remainder operator

For an operation of the form `x % y`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined 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. The remainder operator never causes an overflow.

- 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. 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 same as the scale of `y`, 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`.

14.7.4 Addition operator

For an operation of the form `x + y`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined 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`. If `x + y` is too small to represent in the destination type, `z` is a zero with the same sign as `x + y`.

	y	+0	–0	+∞	–∞	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);
```

The operators are evaluated exactly as `(E)((U)x + (U)y)`.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

The binary + operator performs string concatenation when one or both operands are of type `string`. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted. [*Example:*

```
using System;
class Test
{
   static void Main() {
      string s = null;
      Console.WriteLine("s = >" + s + "<"); // displays s = ><
      int i = 1;
      Console.WriteLine("i = " + i);        // displays i = 1
      float f = 1.2300E+15F;
      Console.WriteLine("f = " + f);        // displays f = 1.23E+15
      decimal d = 2.900m;
      Console.WriteLine("d = " + d);        // displays d = 2.900
   }
}
```

end example]

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

- 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 compile-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. [*Note:* For examples of delegate combination, see §14.7.5 and §22.3. Since `System.Delegate` is not a delegate type, `operator +` is not defined for it. *end note*]

14.7.5 Subtraction operator

For an operation of the form `x – y`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined 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`. If `x – y` is too small to represent in the destination type, `z` is a zero with the same sign as `x – y`.

	y	+0	–0	+∞	–∞	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 compile-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 (§22.1) 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 subset of the first's. (For determining subset equality, corresponding entries are compared as for the delegate equality operator (§14.9.8).) 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 subsets of contiguous entries in the first operand's list, the right-most matching subset of contiguous entries is removed. If removal results in an empty list, the result is `null`. [*Example:* For example:

  ```
  using System;
  delegate void D(int x);
  class Test
  {
     public static void M1(int i) { /* … */ }
     public static void M2(int i) { /* … */ }
  }

  class Demo
  {
     static void Main() {
        D cd1 = new D(Test.M1);
        D cd2 = new D(Test.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
     }
  }
  ```

end example]

14.8 Shift operators

The << and >> operators are used to perform bit shifting operations.

shift-expression:
  additive-expression
  shift-expression << *additive-expression*
  shift-expression >> *additive-expression*

For an operation of the form `x << count` or `x >> count`, binary operator overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

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`.

14.9 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 §14.9.9 and the `as` operator is described in §14.9.10.

The ==, !=, <, >, <= and >= operators are ***comparison operators***. For an operation of the form `x` *op* `y`, where *op* is a comparison operator, overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined 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

14.9.1 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`.

14.9.2 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. [*Example:* For example, if either of `x` and `y` is NaN, then `x < y` is `false`, but `!(x >= y)` is `true`. *end example*]
- When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

  ```
  –∞ < –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.

14.9.3 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`.

14.9.4 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.

14.9.5 Enumeration comparison operators

Every enumeration type implicitly provides the following predefined comparison operators:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating `x` *op* `y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and *op* is one of the comparison operators, is exactly the same as evaluating `((U)x)` *op* `((U)y)`. In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

14.9.6 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 the operands to be *reference-type* values or the value `null`; furthermore, they require that a standard implicit conversion (§13.3.1) exists from the type of either operand to the type of the other operand. Unless both of these conditions are true, a compile-time error occurs. [*Note:* Notable implications of these rules are:

- It is a compile-time error to use the predefined reference type equality operators to compare two references that are known to be different at compile-time. For example, if the compile-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 compile-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.

end note]

For an operation of the form `x == y` or `x != y`, if any applicable `operator` == or `operator` != exists, the operator overload resolution (§14.2.4) 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`. [*Example:* The example

```
using System;
class Test
{
   static void Main() {
      string s = "Test";
      string t = string.Copy(s);
      Console.WriteLine(s == t);
      Console.WriteLine((object)s == t);
      Console.WriteLine(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 (§14.9.7) 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. *end example*]

14.9.7 String equality operators

The predefined string equality operators are: :

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two `string` values are considered equal when one of the following is true:

- Both values are `null`.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string *values* rather than string *references*. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. [*Note:* As described in §14.9.6, the reference type equality operators can be used to compare string references instead of string values. *end note*]

14.9.8 Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators: :

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

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 either of the delegate instances has an invocation list (§22.1) containing one entry, they are equal if and only if the other also has an invocation list containing one entry, and either:
  - Both refer to the same static method, or
  - Both refer to the same non-static method on the same target object.
- If either of the delegate instances has an invocation list containing two or more entries, 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 to the corresponding entry, in order, in the other's invocation list.

Note that delegates of different types can be considered equal by the above definition, as long as they have the same return type and parameter types.

14.9.9 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:

- If the compile-time type of `e` is the same as `T`, or if an implicit reference conversion (§13.1.4) or boxing conversion (§13.1.5) exists from the compile-time type of `e` to `T`:
  - If `e` is of a reference type, the result of the operation is equivalent to evaluating `e != null`.
  - If `e` is of a value type, the result of the operation is `true`.
- Otherwise, if an explicit reference conversion (§13.2.3) or unboxing conversion (§13.2.4) exists from the compile-time type of `e` to `T`, a dynamic type check is performed:
  - If the value of `e` is `null`, the result is `false`.
  - Otherwise, let `R` be the run-time type of the instance referenced by `e`. If `R` and `T` are the same type, if `R` is a reference type and an implicit reference conversion from `R` to `T` exists, or if `R` is a value type and `T` is an interface type that is implemented by `R`, the result is `true`.
  - Otherwise, the result is `false`.
- Otherwise, no reference or boxing conversion of `e` to type `T` is possible, and the result of the operation is `false`.

Note that the `is` operator only considers reference conversions, boxing conversions, and unboxing conversions. Other conversions, such as user defined conversions, are not considered by the `is` operator.

14.9.10 The `as` operator

The `as` operator is used to explicitly convert a value to a given reference type using a reference conversion or a boxing conversion. Unlike a cast expression (§14.6.6), 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. The type of the result is `T`, and the result is always classified as a value. The operation is evaluated as follows:

- If the compile-time type of `e` is the same as `T`, the result is simply the value of `e`.
- Otherwise, if an implicit reference conversion (§13.1.4) or boxing conversion (§13.1.5) exists from the compile-time type of e to `T`, this conversion is performed and becomes the result of the operation.
- Otherwise, if an explicit reference conversion (§13.2.3) exists from the compile-time type of `e` to `T`, a dynamic type check is performed:
  - If the value of e is `null`, the result is the value `null` with the compile-time type `T`.

  - Otherwise, let `R` be the run-time type of the instance referenced by `e`. If `R` and `T` are the same type, if `R` is a reference type and an implicit reference conversion from `R` to `T` exists, or if `R` is a value type and `T` is an interface type that is implemented by `R`, the result is the reference given by `e` with the compile-time type `T`.
  - Otherwise, the result is the value `null` with the compile-time type `T`.
- Otherwise, the indicated conversion is never possible, and a compile-time error occurs.

Note that the `as` operator only performs reference conversions and boxing conversions. Other conversions, such as user defined conversions, are not possible with the `as` operator and should instead be performed using cast expressions.

14.10 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*

For an operation of the form `x` *op* `y`, where *op* is one of the logical operators, overload resolution (§14.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

14.10.1 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.

14.10.2 Enumeration logical operators

Every enumeration type `E` implicitly provides the following predefined logical operators:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

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.

14.10.3 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.

14.11 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 `true`.
- The operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is `false`.

An operation of the form `x && y` or `x || y` is processed by applying overload resolution (§14.2.4) 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 compile-time error occurs.
- Otherwise, if the selected operator is one of the predefined boolean logical operators (§14.10.2), the operation is processed as described in §14.11.1.
- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in §14.11.2.

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 §14.11.2.

14.11.1 Boolean conditional logical operators

When the operands of && or `||` are of type `bool`, or when the operands are of types that do not define an applicable `operator &` or `operator |`, but do define implicit conversions to `bool`, the operation is processed as follows:

- The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.
- The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation.

14.11.2 User-defined conditional logical operators

When the operands of && or `||` are of types that declare an applicable user-defined `operator &` or `operator |`, both of the following must be true, where `T` is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator must be `T`. In other words, the operator must compute the logical `AND` or the logical `OR` of two operands of type `T`, and must return a result of type `T`.
- `T` must contain declarations of `operator true` and `operator false`.

A compile-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 §18.4.2.

14.12 Conditional operator

The `?:` operator is called the conditional operator. It is at times also called the ternary operator.

conditional-expression:
  conditional-or-expression
  conditional-or-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 of a type 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 of the `?:` operator control the type of the conditional expression. Let `X` and `Y` be the types of the second and third operands. Then,

- If `X` and `Y` are the same type, then this is the type of the conditional expression.
- Otherwise, if an implicit conversion (§13.1) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of the conditional expression.
- Otherwise, if an implicit conversion (§13.1) 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.

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.

14.13 Assignment operators

The assignment operators assign a new value to a variable, a property, event, or an indexer element.

assignment:
unary-expression assignment-operator expression

assignment-operator: one of
`=  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=`

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 §17.7.1). The simple assignment operator is described in §14.13.1.

The operators formed by prefixing a binary operator with an `=` character 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 §14.13.2.

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 §14.13.3.

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)`.

14.13.1 Simple assignment

The `=` operator is called the simple assignment operator. In a simple assignment, the right operand must be an expression of a type 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 compile-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 (§13.1).
  - 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 (§13.1.4) 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 (§13.1).
  - The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

[*Note:* The array covariance rules (§19.5) 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[]`. *end note*]

When a property or indexer declared in a *struct-type* is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. [*Note:* Because of §14.5.4, the same rule also applies to fields. e*nd note*]

[*Example:* Given the declarations:

```
struct Point
{
   int x, y;

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

   public int X {
      get { return x; }
      set { x = value; }
   }
```

```
        public int Y {
            get { return y; }
            set { y = value; }
        }
    }

    struct Rectangle
    {
        Point a, b;

        public Rectangle(Point a, Point b) {
            this.a = a;
            this.b = b;
        }

        public Point A {
            get { return a; }
            set { a = value; }
        }

        public Point B {
            get { return b; }
            set { b = value; }
        }
    }
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid, since `r.A` and `r.B` are not variables. *end example*]

14.13.2 Compound assignment

An operation of the form `x` *op*= `y` is processed by applying binary operator overload resolution (§14.2.4) 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`, 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 compile-time error occurs.

The term "evaluated only once" means that in the evaluation of `x` *op* `y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. [*Example:* 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`. *end example*]

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 compile-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 §14.2.6.2. 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. [*Example:* 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. *end example*]

14.13.3 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 (§13.1).
- 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* (§15.6).

14.14 Expression

An *expression* is either a *conditional-expression* or an *assignment*.

> *expression:*
>   *conditional-expression*
>   *assignment*

14.15 Constant expressions

A *constant-expression* is an expression that can be fully evaluated at compile-time.

> *constant-expression:*
>   *expression*

The type of a constant expression can be one of the following: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, any enumeration type, or the null type. 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.

- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- The predefined +, −, !, and ~ unary operators.
- The predefined +, −, *, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <=, and >= binary operators, provided each operand is of a type listed above.
- The ?: conditional operator.

Whenever an expression is of one of the types listed above and contains only the constructs 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 (§14.5.12).

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 (§17.3).
- Enumeration member declarations (§21.30).
- `case` labels of a `switch` statement (§15.7.2).
- `goto case` statements (§15.9.3).
- Dimension lengths in an array creation expression (§14.5.10.2) that includes an initializer.
- Attributes (§24).

An implicit constant expression conversion (§13.1.6) 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.

14.16 Boolean expressions

A *boolean-expression* is an expression that yields a result of type `bool`.

boolean-expression:
  expression

The controlling conditional expression of an *if-statement* (§15.7.1), *while-statement* (§15.8.1), *do-statement* (§15.8.2), or *for-statement* (§15.8.3) is a *boolean-expression*. The controlling conditional expression of the `?:` operator (§14.12) follows the same rules as a *boolean-expression*, but for reasons of operator precedence is classified as a *conditional-or-expression*.

A *boolean-expression* is required to be of a type that can be implicitly converted to `bool` or of a type that implements `operator true`. [*Note:* As required by §17.9.1, any type that implements `operator true` must also implement `operator false`. *end note*] If neither requirement is satisfied, a compile-time error occurs.

When a boolean expression is of a type that cannot be implicitly converted to `bool` but does implement `operator true`, then following evaluation of the expression, the `operator true` implementation provided by that type is invoked to produce a `bool` value.

[*Note:* The `DBBool` struct type in §18.4.2 provides an example of a type that implements `operator true` and `operator false`. *end note*]

15. Statements

C# provides a variety of statements. [*Note:* Most of these statements will be familiar to developers who have programmed in C and C++. *end note*]

statement:
  labeled-statement
  declaration-statement
  embedded-statement

embedded-statement:
  block
  empty-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  try-statement
  checked-statement
  unchecked-statement
  lock-statement
  using-statement

The *embedded-statement* nonterminal is used for statements that appear within other statements. The use of *embedded-statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. [*Example:* The code

```
void F(bool b) {
   if (b)
      int i = 44;
}
```

results in a compile-time error because an `if` statement requires an *embedded-statement* rather than a *statement* for its if branch. If this code were permitted, then the variable `i` would be declared, but it could never be used. (Note, however, that by placing `i`'s declaration in a block, the example is valid.) *end example*]

15.1 End points and reachability

Every statement has an ***end point***. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be ***reachable***. Conversely, if there is no possibility that a statement will be executed, the statement is said to be ***unreachable***.

[*Example:* 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. *end example*]

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

[*Note:* To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (§14.15) that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
   const int i = 1;
   if (i == 2) Console.WriteLine("unreachable");
}
```

the boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. As the constant expression is evaluated at compile-time, producing the value `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a local variable

```
void F() {
   int i = 1;
   if (i == 2) Console.WriteLine("reachable");
}
```

the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be executed. *end note*]

The *block* of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

[*Example:* In the 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 (§15.2).
- The end point of the first `Console.WriteLine` expression statement is reachable15.2 because that statement is reachable (§15.6 and §15.2).
- The `if` statement is reachable because the end point of the first `Console.WriteLine` expression statement is reachable (§15.6 and §15.2).
- The second `Console.WriteLine` expression statement is reachable because the boolean expression of the `if` statement does not have the constant value `false`.

end example]

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the `switch` statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it typically is an indication that a `return` statement is missing.

15.2 Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

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

A *block* consists of an optional *statement-list* (§15.2.1), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements (§15.5). 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 (§14.5.2.1).

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.

15.2.1 Statement lists

A ***statement list*** consists of one or more statements written in sequence. Statement lists occur in *block*s (§15.2) and in *switch-block*s (§15.7.2).

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.

15.3 The empty statement

An *empty-statement* does nothing.

empty-statement:
    `;`

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

[*Example:* An empty statement can be used when writing a `while` statement with a null body:

```
bool ProcessMessage() {…}
```

```
void ProcessMessages() {
   while (ProcessMessage())
      ;
}
```

Also, an empty statement can be used to declare a label just before the closing "`}`" of a block:

```
void F() {
   …
   if (done) goto exit;
   …
   exit: ;
}
```

end example]

15.4 Labeled statements

A *labeled-statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

labeled-statement:
  identifier `:` *statement*

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from `goto` statements (§15.9.3) within the scope of the label. [*Note:* This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks. *end note*]

Labels have their own declaration space and do not interfere with other identifiers. [*Example:* The example

```
int F(int x) {
   if (x >= 0) goto x;
   x = -x;
   x: return x;
}
```

is valid and uses the name `x` as both a parameter and a label. e*nd example*]

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable `goto` statement. (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.)

15.5 Declaration statements

A *declaration-statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

declaration-statement:
  local-variable-declaration `;`
  local-constant-declaration `;`

15.5.1 Local variable declarations

A *local-variable-declaration* declares one or more local variables.

local-variable-declaration:
  type *local-variable-declarators*

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 *type* of a *local-variable-declaration* specifies the type of the variables introduced by the declaration. The type is followed by a list of *local-variable-declarator*s, each of which introduces a new variable. A *local-variable-declarator* consists of an *identifier* that names the variable, optionally followed by an "=" token and a *local-variable-initializer* that gives the initial value of the variable.

The value of a local variable is obtained in an expression using a *simple-name* (§14.5.2), and the value of a local variable is modified using an *assignment* (§14.13). A local variable must be definitely assigned (§12.3) 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.

[*Example:* The example

```
void F() {
   int x = 1, y, z = x * 2;
}
```

corresponds exactly to

```
void F() {
   int x; x = 1;
   int y;
   int z; z = x * 2;
}
```

end example]

15.5.2 Local constant declarations

A *local-constant-declaration* declares one or more local constants.

local-constant-declaration:
`const` *type* *constant-declarators*

constant-declarators:
constant-declarator
constant-declarators **,** *constant-declarator*

constant-declarator:
identifier = *constant-expression*

The *type* of a *local-constant-declaration* specifies the type of the constants introduced by the declaration. The type is followed by a list of *constant-declarator*s, each of which introduces a new constant. A *constant-declarator* consists of an *identifier* that names the constant, followed by an "=" token, followed by a *constant-expression* (§14.15) 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 (§17.3).

The value of a local constant is obtained in an expression using a *simple-name* (§14.5.2).

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.

15.6 Expression statements

An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement:
  statement-expression `;`

statement-expression:
  invocation-expression
  object-creation-expression
  assignment
  post-increment-expression
  post-decrement-expression
  pre-increment-expression
  pre-decrement-expression

Not all expressions are permitted as statements. [*Note:* In particular, expressions such as `x + y` and `x == 1`, that merely compute a value (which will be discarded), are not permitted as statements. *end note*]

Execution of an expression statement evaluates the contained expression and then transfers control to the end point of the expression statement. The end point of an *expression-statement* is reachable if that *expression-statement* is reachable.

15.7 Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

selection-statement:
  if-statement
  switch-statement

15.7.1 The `if` statement

The `if` statement selects a statement for execution based on the value of a boolean expression.

if-statement:
  `if ( ` *boolean-expression* ` )` *embedded-statement*
  `if ( ` *boolean-expression* ` )` *embedded-statement* `else` *embedded-statement*

boolean-expression:
  expression

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. [*Example:* Thus, an `if` statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {
   if (y) {
      F();
   }
   else {
      G();
   }
}
```

end example]

An `if` statement is executed as follows:

- The *boolean-expression* (§14.16) 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`.

15.7.2 The `switch` statement

The switch statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

switch-statement:
  `switch ( ` *expression* ` )` *switch-block*

switch-block:
  `{` *switch-sections*$_{opt}$ `}`

switch-sections:
  switch-section
  switch-sections switch-section

switch-section:
  switch-labels statement-list

switch-labels:
  switch-label
  switch-labels switch-label

switch-label:
  `case` *constant-expression* `:`
  `default :`

A *switch-statement* consists of the keyword `switch`, followed by a parenthesized expression (called the switch expression), followed by a *switch-block*. The *switch-block* consists of zero or more *switch-section*s, enclosed in braces. Each *switch-section* consists of one or more *switch-labels* followed by a *statement-list* (§15.2.1).

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`, `char`, `string`, or an *enum-type*, then that is the governing type of the `switch` statement. Otherwise, exactly one user-defined implicit conversion (§13.4) 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`. 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 of a type that is implicitly convertible (§13.1) 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. [*Example:* The example

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
default:
   CaseOthers();
   break;
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to “fall through” to the next switch section, and the example

```
switch (i) {
case 0:
   CaseZero();
case 1:
   CaseZeroOrOne();
default:
   CaseAny();
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit `goto case` or `goto default` statement must be used:

```
switch (i) {
case 0:
   CaseZero();
   goto case 1;
case 1:
   CaseZeroOrOne();
   goto default;
default:
   CaseAny();
   break;
}
```

end example]

Multiple labels are permitted in a *switch-section*. [*Example:* The example

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
case 2:
default:
   CaseTwo();
   break;
}
```

is valid. The example does not violate the "no fall through" rule because the labels `case 2:` and `default:` are part of the same *switch-section*. *end example*]

[*Note:* The "no fall through" rule prevents a common class of bugs that occur in C and C++ when `break` statements are accidentally omitted. 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;
}
```

end note]

[*Note:* The statement list of a switch section typically ends in a `break`, `goto case`, or `goto default` statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a `while` statement controlled by the boolean expression `true` is known to never reach its end point. Likewise, a `throw` or `return` statement always transfers control elsewhere and never reaches its end point. Thus, the following example is valid:

```
switch (i) {
case 0:
   while (true) F();
case 1:
   throw new ArgumentException();
case 2:
   return;
}
```

end note]

[*Example:* The governing type of a `switch` statement 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;
   }
}
```

end example]

[*Note:* Like the string equality operators (§14.9.7), 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. *end note*]

When the governing type of a `switch` statement is `string`, the value `null` is permitted as a case label constant.

The *statement-list*s of a *switch-block* may contain declaration statements (§15.5). 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 (§14.5.2.1).

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.

15.8 Iteration statements

Iteration statements repeatedly execute an embedded statement.

iteration-statement:
  while-statement
  do-statement
  for-statement
  foreach-statement

15.8.1 The `while` statement

The `while` statement conditionally executes an embedded statement zero or more times.

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

A `while` statement is executed as follows:

- The *boolean-expression* (§14.16) 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 (§15.9.1) 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 (§15.9.2) 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`.

15.8.2 The `do` statement

The `do` statement conditionally executes an embedded statement one or more times.

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

A `do` statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the *boolean-expression* (§14.16) 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 (§15.9.1) 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 (§15.9.2) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the `do` statement).

The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

The end point of a `do` statement is reachable if at least one of the following is true:

- The `do` statement contains a reachable `break` statement that exits the `do` statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value `true`.

15.8.3 The `for` statement

The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

for-statement:
`for ( `*for-initializer*$_{opt}$` ; `*for-condition*$_{opt}$` ; `*for-iterator*$_{opt}$` ) `*embedded-statement*

for-initializer:
local-variable-declaration
statement-expression-list

for-condition:
boolean-expression

for-iterator:
statement-expression-list

statement-expression-list:
statement-expression
statement-expression-list `,` *statement-expression*

The *for-initializer*, if present, consists of either a *local-variable-declaration* (§15.5.1) or a list of *statement-expression*s (§15.6) separated by commas. The scope of a local variable declared by a *for-initializer* starts at the *local-variable-declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for-condition* and the *for-iterator*.

The *for-condition*, if present, must be a *boolean-expression* (§14.16).

The *for-iterator*, if present, consists of a list of *statement-expression*s (§15.6) 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 (§15.9.1) may be used to transfer control to the end point of the `for` statement (thus ending iteration of the embedded statement), and a `continue` statement (§15.9.2) may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the `for` statement).

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`.

15.8.4 The `foreach` statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement:
`foreach` `(` *type* *identifier* `in` *expression* `)` *embedded-statement*

The *type* and *identifier* of a `foreach` statement declare the ***iteration variable*** of the statement. 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.

The type of the *expression* of a `foreach` statement must be a collection type (as defined below), and an explicit conversion (§13.2) must exist from the element type of the collection to the type of the iteration variable. If *expression* has the value `null`, a `System.NullReferenceException` is thrown.

A type `C` is said to be a ***collection type*** if it implements the `System.IEnumerable` interface or implements the ***collection pattern*** by meeting all of the following criteria:

- `C` contains a `public` instance method with the signature `GetEnumerator()`, that returns a *struct-type*, *class-type*, or *interface-type*, which is called `E` in the following text.
- `E` contains a `public` instance method with the signature `MoveNext()` and the return type `bool`.
- `E` contains a `public` instance property named `Current` that permits reading the current value. The type of this property is said to be the ***element type*** of the collection type.

A type that implements `IEnumerable` is also a collection type, even if it doesn't satisfy the conditions above. (This is possible if it implements `IEnumerable` via private interface implementation.)

The `System.Array` type (§19.1.1) is a collection type, and since all array types derive from `System.Array`, any array type expression is permitted in a `foreach` statement. 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.

A `foreach` statement of the form:

```
foreach (ElementType element in collection) statement
```

corresponds to one of two possible expansions:

- If the `collection` expression is of a type that implements the collection pattern (as defined above), the expansion of the `foreach` statement is:

```
Enumerator enumerator = (collection).GetEnumerator();
try {
   while (enumerator.MoveNext()) {
      ElementType element = (ElementType)enumerator.Current;
      statement;
   }
}
finally {
   IDisposable disposable = enumerator as System.IDisposable;
   if (disposable != null) disposable.Dispose();
}
```

  [*Note*: Significant optimizations of the above are often easily available. If the type `E` implements `System.IDisposable`, then the expression `(enumerator as System.IDisposable)` will always be non-null and the implementation can safely substitute a simple conversion for a possibly more expensive type test. Conversely, if the type `E` is `sealed` and does not implement `System.IDisposable`, then the expression `(enumerator as System.IDisposable)` will always evaluate to null. In this case, the implementation can safely optimize away the entire finally clause. *end note*]

- Otherwise; the `collection` expression is of a type that implements `System.IEnumerable`, and the expansion of the `foreach` statement is:

```
IEnumerator enumerator =
((System.IEnumerable)(collection)).GetEnumerator();
try {
   while (enumerator.MoveNext()) {
      ElementType element = (ElementType)enumerator.Current;
      statement;
   }
}
finally {
   IDisposable disposable = enumerator as System.IDisposable;
   if (disposable != null) disposable.Dispose();
}
```

In either expansion, the `enumerator` variable is a temporary variable that is inaccessible in, and invisible to, the embedded `statement`, and the `element` variable is read-only in the embedded `statement`.

[*Example:* The following example prints out each value in a two-dimensional array, in element order:

```
using System;
class Test
{
   static void Main() {
      double[,] values = {
         {1.2, 2.3, 3.4, 4.5},
         {5.6, 6.7, 7.8, 8.9}
      };

      foreach (double elementValue in values)
         Console.Write("{0} ", elementValue);
      Console.WriteLine();
   }
}
```

The output produced is as follows:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

end example]

15.9 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.

[*Example:* 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
```

end example]

15.9.1 The `break` statement

The `break` statement exits the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement.

break-statement:
`break ;`

The target of a `break` statement is the end point of the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement. If a `break` statement is not enclosed by a `switch`, `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `switch`, `while`, `do`, `for`, or `foreach` statements are nested within each other, a `break` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (§15.9.3) must be used.

A `break` statement cannot exit a `finally` block (§15.10). 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.

15.9.2 The `continue` statement

The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

continue-statement:
`continue ;`

The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (§15.9.3) must be used.

A `continue` statement cannot exit a `finally` block (§15.10). 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.

15.9.3 The `goto` statement

The `goto` statement transfers control to a statement that is marked by a label.

goto-statement:
`goto` *identifier* `;`
`goto case` *constant-expression* *;*
`goto default ;`

The target of a `goto` *identifier* statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. [*Note:* This rule permits the use of a `goto` statement to transfer control *out of* a nested scope, but not *into* a nested scope. In the example

```
using System;
class Test
{
   static void Main(string[] args) {
      string[,] table = {
         {"red", "blue", "green"},
         {"Monday", "Wednesday", "Friday"}
      };

      foreach (string str in args) {
         int row, colm;
         for (row = 0; row <= 1; ++row)
            for (colm = 0; colm <= 2; ++colm)
               if (str == table[row,colm])
                  goto done;
```

```
                Console.WriteLine("{0} not found", str);
                continue;
            done:
                Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
            }
        }
    }
```

a `goto` statement is used to transfer control out of a nested scope. *end note*]

The target of a `goto case` statement is the statement list in the immediately enclosing switch statement (§15.7.2) which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the *constant-expression* is not implicitly convertible (§13.1) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.

The target of a `goto default` statement is the statement list in the immediately enclosing switch statement (§15.7.2), which contains a `default` label. If the `goto default` statement is not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.

A `goto` statement cannot exit a `finally` block (§15.10). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.

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.

15.9.4 The `return` statement

The `return` statement returns control to the caller of the function member in which the `return` statement appears.

return-statement:
`return` *expression*$_{opt}$ `;`

A `return` statement with no expression can be used only in a function member that does not compute a value; that is, a method with the return type `void`, the `set` accessor of a property or indexer, the add and remove accessors of an event, an instance constructor, static constructor, or a destructor.

A `return` statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void return type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion (§13.1) must exist from the type of the expression to the return type of the containing function member.

It is a compile-time error for a `return` statement to appear in a `finally` block (§15.10).

A `return` statement is executed as follows:

- If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.

- If the `return` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the containing function member.

Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return` statement is never reachable.

15.9.5 The `throw` statement

The `throw` statement throws an exception.

throw-statement:
`throw` *expression*$_{opt}$ `;`

A `throw` statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type `System.Exception` or of a class type that derives from `System.Exception`. 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` (§15.10) clause is considered a match for any exception type. If a matching `catch` clause is located, the exception propagation is completed by transferring control to the block of that `catch` clause.
    - Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the `finally` block, processing of the current exception is continued.
- If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

15.10 The `try` statement

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when control leaves the `try` statement.

try-statement:
  `try` *block catch-clauses*
  `try` *block finally-clause*
  `try` *block catch-clauses finally-clause*

catch-clauses:
  specific-catch-clauses general-catch-clause$_{opt}$
  specific-catch-clauses$_{opt}$ *general-catch-clause*

specific-catch-clauses:
  specific-catch-clause
  specific-catch-clauses specific-catch-clause

specific-catch-clause:
  `catch ( ` *class-type identifier*$_{opt}$ `)` *block*

general-catch-clause:
  `catch` *block*

finally-clause:
  `finally` *block*

There are three possible forms of `try` statements:

- A `try` block followed by one or more `catch` blocks.
- A `try` block followed by a `finally` block.
- A `try` block followed by one or more `catch` blocks followed by a `finally` block.

When a `catch` clause specifies a *class-type*, the type must be `System.Exception` or a type that derives from `System.Exception`.

When a `catch` clause specifies both a *class-type* and an *identifier*, an ***exception variable*** of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the `catch` block. During execution of the `catch` block, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a `catch` clause includes an exception variable name, it is impossible to access the exception object in the `catch` block.

A `catch` clause that specifies neither an exception type nor an exception variable name is called a general `catch` clause. A `try` statement can only have one general `catch` clause, and if one is present it must be the last `catch` clause.

[*Note:* Some environments, especially those supporting multiple languages, may support exceptions that are not representable as an object derived from `System.Exception`, although such an exception could never be generated by C# code. In such an environment, a general catch clause might be used to catch such an exception. 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. *end note*]

In order to locate a handler for an exception, `catch` clauses are examined in lexical order. A compile-time error occurs if a `catch` clause specifies a type that is the same as, or is derived from, a type that was specified in an earlier `catch` clause for the same `try`. [*Note:* Without this restriction, it would be possible to write unreachable `catch` clauses. *end note*]

Within a `catch` block, a `throw` statement (§15.9.5) 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.

[*Example:* 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 would be as follows:

```
Exception in F: G
Exception in Main: F
```

end example]

It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally` block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement 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, 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 (§15.9.5).

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.

15.11 The `checked` and `unchecked` statements

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

checked-statement:
  `checked` *block*

unchecked-statement:
  `unchecked` *block*

The `checked` statement causes all expressions in the *block* to be evaluated in a checked context, 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 (§14.5.12), except that they operate on blocks instead of expressions.

15.12 The `lock` statement

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

lock-statement:
    `lock` `(` *expression* `)` *embedded-statement*

The expression of a `lock` statement must denote a value of a *reference-type*. No implicit boxing conversion (§13.1.5) 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

```
System.Threading.Monitor.Enter(x);
try {
   …
}
finally {
   System.Threading.Monitor.Exit(x);
}
```

except that `x` is only evaluated once.

[*Example:* The `System.Type` object of a class can conveniently be used as the mutual-exclusion lock for static methods of the class. For example:

```
class Cache
{
   public static void Add(object x) {
      lock (typeof(Cache)) {
         …
      }
   }
   public static void Remove(object x) {
      lock (typeof(Cache)) {
         …
      }
   }
}
```

end example]

15.13 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 `System.IDisposable` or a type that can be implicitly converted to `System.IDisposable`. If the form of *resource-acquisition* is *expression* then this expression must be `System.IDisposable` or a type that can be implicitly converted 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) 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.

A `using` statement of the form

```
using (R r1 = new R()) {
   r1.F();
}
```

is precisely equivalent to

```
R r1 = new R();
try {
   r1.F();
}
finally {
   if (r1 != null) ((IDisposable)r1).Dispose();
}
```

A *resource-acquisition* may acquire multiple resources of a given type. This is equivalent to nested `using` statements. A using statement of the form

```
using (R r1 = new R(), r2 = new R()) {
   r1.F();
   r2.F();
}
```

is precisely equivalent to:

```
using (R r1 = new R())
   using (R r2 = new R()) {
      r1.F();
      r2.F();
   }
```

which is, by expansion, precisely equivalent to:

```
R r1 = new R();
try {
   R r2 = new R();
   try {
      r1.F();
      r2.F();
   }
   finally {
      if (r2 != null) ((IDisposable)r2).Dispose();
   }
}
finally {
   if (r1 != null) ((IDisposable)r1).Dispose();
}
```

16. Namespaces

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Using-directive*s* (§16.3) are provided to facilitate the use of namespaces.

16.1 Compilation units

A *compilation-unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using-directive*s followed by zero or more *global-attributes* followed by zero or more *namespace-member-declaration*s.

> *compilation-unit:*
> *using-directives*$_{opt}$ *global-attributes*$_{opt}$ *namespace-member-declarations*$_{opt}$

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *using-directives* of a compilation unit affect the *global-attributes* and *namespace-member-declarations* of that compilation unit, but have no effect on other compilation units.

The *global-attributes* (§24) of a compilation unit permit the specification of attributes for the target assembly. Assemblies act as physical containers for types.

The *namespace-member-declarations* of each compilation unit of a program contribute members to a single declaration space called the global namespace. [*Example:* For example:

File `A.cs`:

```
class A {}
```

File `B.cs`:

```
class B {}
```

The two compilation units contribute to the single global namespace, in this case declaring two classes with the fully qualified names `A` and `B`. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name. *end example*]

16.2 Namespace declarations

A *namespace-declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon.

> *namespace-declaration:*
> `namespace` *qualified-identifier* *namespace-body* `;`$_{opt}$
>
> *qualified-identifier:*
> *identifier*
> *qualified-identifier* `.` *identifier*
>
> *namespace-body:*
> `{` *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ `}`

A *namespace-declaration* may occur as a top-level declaration in a *compilation-unit* or as a member declaration within another *namespace-declaration*. When a *namespace-declaration* occurs as a top-level declaration in a *compilation-unit*, the namespace becomes a member of the global namespace. When a *namespace-declaration* occurs within another *namespace-declaration*, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace must be unique within the containing namespace.

Namespaces are implicitly `public` and the declaration of a namespace cannot include any access modifiers.

Within a *namespace-body*, the optional *using-directives* import the names of other namespaces and types, allowing them to be referenced directly instead of through qualified names. The optional *namespace-member-declarations* contribute members to the declaration space of the namespace. Note that all *using-directives* must appear before any member declarations.

The *qualified-identifier* of a *namespace-declaration* may be a single identifier or a sequence of identifiers separated by "`.`" tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. [*Example:* For example,

```
namespace N1.N2
{
   class A {}
   class B {}
}
```

is semantically equivalent to

```
namespace N1
{
   namespace N2
   {
      class A {}
      class B {}
   }
}
```

end example]

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space (§10.3). [*Example:* In the example

```
namespace N1.N2
{
   class A {}
}
namespace N1.N2
{
   class B {}
}
```

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name. *end example*]

16.3 Using directives

Using-directives facilitate the use of namespaces and types defined in other namespaces. *Using-directives* impact the name resolution process of *namespace-or-type-name*s (§10.8) and *simple-name*s (§14.5.2), but unlike declarations, *using-directives* do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

using-directives:
  using-directive
  using-directives   using-directive

using-directive:
  using-alias-directive
  using-namespace-directive

A *using-alias-directive* (§16.3.1) introduces an alias for a namespace or type.

A *using-namespace-directive* (§16.3.2) imports the type members of a namespace.

The scope of a *using-directive* extends over the *namespace-member-declarations* of its immediately containing compilation unit or namespace body. The scope of a *using-directive* specifically does not include its peer *using-directive*s. Thus, peer *using-directive*s do not affect each other, and the order in which they are written is insignificant.

16.3.1 Using alias directives

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

using-alias-directive:
  `using` *identifier* `=` *namespace-or-type-name* `;`

Within member declarations in a compilation unit or namespace body that contains a *using-alias-directive*, the identifier introduced by the *using-alias-directive* can be used to reference the given namespace or type. [*Example:* For example:

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using A = N1.N2.A;
   class B: A {}
}
```

Above, within member declarations in the `N3` namespace, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

```
namespace N3
{
   using R = N1.N2;
   class B: R.A {}
}
```

end example]

The *identifier* of a *using-alias-directive* must be unique within the declaration space of the compilation unit or namespace that immediately contains the *using-alias-directive*. [*Example:* For example:

```
namespace N3
{
   class A {}
}
namespace N3
{
   using A = N1.N2.A;      // Error, A already exists
}
```

Above, `N3` already contains a member A, so it is a compile-time error for a *using-alias-directive* to use that identifier. *end example*] Likewise, it is a compile-time error for two or more *using-alias-directive*s in the same compilation unit or namespace body to declare aliases by the same name.

A *using-alias-directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, a *using-alias-*

directive is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs. [*Example:* In the example

```
namespace N3
{
   using R = N1.N2;
}
namespace N3
{
   class B: R.A {}         // Error, R unknown
}
```

the scope of the *using-alias-directive* that introduces `R` only extends to member declarations in the namespace body in which it is contained, so `R` is unknown in the second namespace declaration. However, placing the *using-alias-directive* in the containing compilation unit causes the alias to become available within both namespace declarations:

```
using R = N1.N2;
namespace N3
{
   class B: R.A {}
}
namespace N3
{
   class C: R.A {}
}
```

end example]

Just like regular members, names introduced by *using-alias-directive*s are hidden by similarly named members in nested scopes. [*Example:* In the example

```
using R = N1.N2;
namespace N3
{
   class R {}
   class B: R.A {}      // Error, R has no member A
}
```

the reference to `R.A` in the declaration of `B` causes a compile-time error because `R` refers to `N3.R`, not `N1.N2`. *end example*]

The order in which *using-alias-directive*s are written has no significance, and resolution of the *namespace-or-type-name* referenced by a *using-alias-directive* is not affected by the *using-alias-directive* itself or by other *using-directive*s in the immediately containing compilation unit or namespace body. In other words, the *namespace-or-type-name* of a *using-alias-directive* is resolved as if the immediately containing compilation unit or namespace body had no *using-directive*s. [*Example:* In the example

```
namespace N1.N2 {}
namespace N3
{
   using R1 = N1;       // OK
   using R2 = N1.N2;    // OK
   using R3 = R1.N2;    // Error, R1 unknown
}
```

the last *using-alias-directive* results in a compile-time error because it is not affected by the first *using-alias-directive*. *end example*]

A *using-alias-directive* can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type nested within that namespace.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name. [*Example:* For example, given

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using R1 = N1;
   using R2 = N1.N2;
   class B
   {
      N1.N2.A a;          // refers to N1.N2.A
      R1.N2.A b;          // refers to N1.N2.A
      R2.A c;             // refers to N1.N2.A
   }
}
```

the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent and all refer to the class whose fully qualified name is `N1.N2.A`. *end example*]

16.3.2 Using namespace directives

A *using-namespace-directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

using-namespace-directive:
   `using` *namespace-name* `;`

Within member declarations in a compilation unit or namespace body that contains a *using-namespace-directive*, the types contained in the given namespace can be referenced directly. [*Example:* For example:

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using N1.N2;
   class B: A {}
}
```

Above, within member declarations in the `N3` namespace, the type members of `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`. *end example*]

A *using-namespace-directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. [*Example:* In the example

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using N1;
   class B: N2.A {}      // Error, N2 unknown
}
```

the *using-namespace-directive* imports the types contained in `N1`, but not the namespaces nested in `N1`. Thus, the reference to `N2.A` in the declaration of `B` results in a compile-time error because no members named `N2` are in scope. *end example*]

Unlike a *using-alias-directive*, a *using-namespace-directive* may import types whose identifiers are already defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using-*

namespace-directive are hidden by similarly named members in the enclosing compilation unit or namespace body. [*Example:* For example:

```
namespace N1.N2
{
   class A {}
   class B {}
}
namespace N3
{
   using N1.N2;
   class A {}
}
```

Here, within member declarations in the `N3` namespace, `A` refers to `N3.A` rather than `N1.N2.A`. *end example*]

When more than one namespace imported by *using-namespace-directive*s in the same compilation unit or namespace body contain types by the same name, references to that name are considered ambiguous. [*Example:* In the example

```
namespace N1
{
   class A {}
}
namespace N2
{
   class A {}
}
namespace N3
{
   using N1;
   using N2;
   class B: A {}           // Error, A is ambiguous
}
```

both `N1` and `N2` contain a member `A`, and because `N3` imports both, referencing `A` in `N3` is a compile-time error. *end example*] In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing a *using-alias-directive* that picks a particular `A`. [*Example:* For example:

```
namespace N3
{
   using N1;
   using N2;
   using A = N1.A;
   class B: A {}           // A means N1.A
}
```

end example]

Like a *using-alias-directive*, a *using-namespace-directive* does not contribute any new members to the underlying declaration space of the compilation unit or namespace, but, rather, affects only the compilation unit or namespace body in which it appears.

The *namespace-name* referenced by a *using-namespace-directive* is resolved in the same way as the *namespace-or-type-name* referenced by a *using-alias-directive*. Thus, *using-namespace-directive*s in the same compilation unit or namespace body do not affect each other and can be written in any order.

16.4 Namespace members

A *namespace-member-declaration* is either a *namespace-declaration* (§16.2) or a *type-declaration* (§16.5).

namespace-member-declarations:
namespace-member-declaration
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
namespace-declaration
type-declaration

A compilation unit or a namespace body can contain *namespace-member-declarations*, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

16.5 Type declarations

A *type-declaration* is a *class-declaration* (§17.1), a *struct-declaration* (§18.1), an *interface-declaration* (§20.1), an *enum-declaration* (§21.1), or a *delegate-declaration* (§22.1).

type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration

A *type-declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

When a type declaration for a type `T` occurs as a top-level declaration in a compilation unit, the fully qualified name of the newly declared type is simply `T`. When a type declaration for a type `T` occurs within a namespace, class, or struct, the fully qualified name of the newly declared type is `N.T`, where `N` is the fully qualified name of the containing namespace, class, or struct.

A type declared within a class or struct is called a nested type (§17.2.6).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (§10.5.1):

- Types declared in compilation units or namespaces can have `public` or `internal` access. The default is `internal` access.
- Types declared in classes can have `public`, `protected internal`, `protected`, `internal`, or `private` access. The default is `private` access.
- Types declared in structs can have `public`, `internal`, or `private` access. The default is `private` access.

17. Classes

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

17.1 Class declarations

A *class-declaration* is a *type-declaration* (§16.5) that declares a new class.

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `class` *identifier* *class-base*$_{opt}$ *class-body* `;`$_{opt}$

A *class-declaration* consists of an optional set of *attributes* (§24), followed by an optional set of *class-modifiers* (§17.1.1), followed by the keyword `class` and an *identifier* that names the class, followed by an optional *class-base* specification (§17.1.2), followed by a *class-body* (§17.1.3), optionally followed by a semicolon.

17.1.1 Class modifiers

A *class-declaration* may optionally include a sequence of class modifiers:

class-modifiers:
class-modifier
class-modifiers *class-modifier*

class-modifier:
`new`
`public`
`protected`
`internal`
`private`
`abstract`
`sealed`

It is a compile-time error for the same modifier to appear multiple times in a class declaration.

The `new` modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name, as described in §10.2.2. 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 (§10.5.1).

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

17.1.1.1 Abstract classes

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An ***abstract class*** differs from a ***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. [*Example:* In the example

```
abstract class A
{
   public abstract void F();
}
abstract class B: A
{
   public void G() {}
}
class C: B
{
   public override void F() {
      // actual implementation of F
   }
}
```

the abstract class `A` introduces an abstract method `F`. Class `B` introduces an additional method `G`, but since it doesn't provide an implementation of `F`, `B` must also be declared abstract. Class `C` overrides `F` and provides an actual implementation. Since there are no abstract members in `C`, `C` is permitted (but not required) to be non-abstract. *end example*]

17.1.1.2 Sealed classes

The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

[*Note:* The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations. *end note*]

17.1.2 Class base specification

A class declaration may include a *class-base* specification, which defines the direct base class of the class and the interfaces (§20) implemented by the class.

class-base:
  `:` *class-type*
  `:` *interface-type-list*
  `:` *class-type* `,` *interface-type-list*

interface-type-list:
  interface-type
  interface-type-list `,` *interface-type*

17.1.2.1 Base classes

When a *class-type* is included in the *class-base*, it specifies the direct base class of the class being declared. If a class declaration has no *class-base*, or if the *class-base* lists only interface types, the direct base class is assumed to be `object`. A class inherits members from its direct base class, as described in §17.2.1.

[*Example:* In the example

```
class A {}
class B: A {}
```

class `A` is said to be the direct base class of `B`, and `B` is said to be derived from `A`. Since `A` does not explicitly specify a direct base class, its direct base class is implicitly `object`. *end example*]

The direct base class of a class type must be at least as accessible as the class type itself (§10.5.4). For example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.

The base classes of a class are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. [*Note:* Referring to the example above, the base classes of `B` are `A` and `object`. *end note*]

Except for class `object`, every class has exactly one direct base class. The `object` class has no direct base class and is the ultimate base class of all other classes.

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

[*Example:* The example

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Likewise, the example

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

results in a compile-time error because `A` depends on `B.C` (its direct base class), which depends on `B` (its immediately enclosing class), which circularly depends on A. *end example*]

Note that a class does not depend on the classes that are nested within it. [*Example:* In the example

```
class A
{
   class B: A {}
}
```

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

It is not possible to derive from a `sealed` class. [*Example:* In the example

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

class `B` results in a compile-time error because it attempts to derive from the `sealed` class A. *end example*]

17.1.2.2 Interface implementations

A *class-base* specification may include a list of interface types, in which case the class is said to implement the given interface types. Interface implementations are discussed further in §20.4.

17.1.3 Class body

The *class-body* of a class defines the members of that class.

class-body:
{ *class-member-declarations*$_{opt}$ }

17.2 Class members

The members of a class consist of the members introduced by its *class-member-declaration*s and the members inherited from the direct base class.

class-member-declarations:
class-member-declaration
class-member-declarations class-member-declaration

class-member-declaration:
constant-declaration
field-declaration
method-declaration
property-declaration
event-declaration
indexer-declaration
operator-declaration
constructor-declaration
destructor-declaration
static-constructor-declaration
type-declaration

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

- Constants, which represent constant values associated with that class (§17.3).
- Fields, which are the variables of that class (§17.4).
- Methods, which implement the computations and actions that can be performed by that class (§17.5).
- Properties, which define named characteristics and the actions associated with reading and writing those characteristics (§17.6).
- Events, which define notifications that can be generated by that class (§17.7).
- Indexers, which permit instances of that class to be indexed in the same way as arrays (§17.8).
- Operators, which define the expression operators that can be applied to instances of that class (§17.9).
- Instance constructors, which implement the actions required to initialize instances of that class (§17.10)
- Destructors, which implement the actions to be performed before instances of that class are permanently discarded (§17.12).
- Static constructors, which implement the actions required to initialize that class itself (§17.11).
- Types, which represent the types that are local to that class (§16.5).

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

A *class-declaration* creates a new declaration space (§10.3), and the *class-member-declarations* immediately contained by the *class-declaration* introduce new members into this declaration space. The following rules apply to *class-member-declaration*s:

- Instance constructors, destructors, and static constructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.

- The 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 (§10.6) of a method must differ from the signatures of all other methods declared in the same class.
- The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class.
- 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 (§17.2.1) are not part of the declaration space of a class. [*Note:* Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member). *end note*]

17.2.1 Inheritance

A class ***inherits*** the members of its direct base class. Inheritance means that a class implicitly contains all members of its direct base class, except for the instance constructors, destructors, and static constructors of the base class. Some important aspects of inheritance are:

- Inheritance is transitive. If `C` is derived from B, and `B` is derived from A, then `C` inherits the members declared in `B` as well as the members declared in A.
- A derived class *extends* its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Instance constructors, destructors, and static constructors are not inherited, but all other members are, regardless of their declared accessibility (§10.5). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
- A derived class can ***hide*** (§10.7.1.2) inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove that member—it merely makes that member inaccessible in the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion (§13.1.4) exists from a derived class type to any of its base class types. Thus, a reference to an instance of some derived class can be treated as a reference to an instance of any of its base classes.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which that function member is invoked.

17.2.2 The `new` modifier

A *class-member-declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to ***hide*** the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in §10.7.1.2.

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

17.2.3 Access modifiers

A *class-member-declaration* can have any one of the five possible kinds of declared accessibility (§10.5.1): `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected internal` combination, it is a compile-time error to specify more than one access modifier. When a *class-member-declaration* does not include any access modifiers, `private` is assumed.

17.2.4 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 (§10.5.4).

17.2.5 Static and instance members

Members of a class are either ***static members*** or ***instance members***. [*Note:* Generally speaking, it is useful to think of static members as belonging to classes and instance members as belonging to objects (instances of classes). *end note*]

When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member is referenced in a *member-access* (§14.5.4) of the form `E.M`, `E` must denote a type that has a member `M`. It is a compile-time error for `E` to denote an instance.
- A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.
- 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 is referenced in a *member-access* (§14.5.4) of the form `E.M`, `E` must denote an instance of a type that has a member `M`. It is a compile-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` (§14.5.7).

[*Example:* 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* (§14.5.2) can be used to access both instance members and static members. The `G` method shows that in a static function member, it is a compile-time error to access an instance member through a *simple-name*. The `Main` method shows that in a *member-access* (§14.5.4), instance members must be accessed through instances, and static members must be accessed through types. *end example*]

17.2.6 Nested types

A type declared within a class or struct is called a ***nested type***. A type that is declared within a compilation unit or namespace is called a ***non-nested type***. [*Example:* In the following example:

```
using System;
class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

class `B` is a nested type because it is declared within class `A`, and class `A` is a non-nested type because it is declared within a compilation unit. *end example*]

17.2.6.1 Fully qualified name

The fully qualified name (§10.8.1) for a nested type is `S.N` where `S` is the fully qualified name of the type in which type `N` is declared.

17.2.6.2 Declared accessibility

Non-nested types can have public or internal declared accessibility and they internal declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of five forms of declared accessibility (public, protected internal, protected, internal, or private) and, like other class members, defaults to private declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (public, internal, or private) and, like other struct members, defaults to private declared accessibility.

[*Example:* The example

```
public class List
{
   // Private data structure
   private class Node
   {
      public object Data;
      public Node Next;
      public Node(object data, Node next) {
         this.Data = data;
         this.Next = next;
      }
   }

   private Node first = null;
   private Node last = null;

   // Public interface
   public void AddToFront(object o) {…}
   public void AddToBack(object o) {…}
   public object RemoveFromFront() {…}
   public object AddToFront() {…}
   public int Count { get {…} }
}
```

declares a private nested class `Node`. *end example*]

17.2.6.3 Hiding

A nested type may hide (§10.7.1.1) a base member. The `new` modifier is permitted on nested type declarations so that hiding can be expressed explicitly. [*Example:* The example

```
using System;
class Base
{
   public static void M() {
      Console.WriteLine("Base.M");
   }
}

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

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

shows a nested class `M` that hides the method `M` defined in `Base`. *end example*]

17.2.6.4 this access

A nested type and its containing type do not have a special relationship with regard to *this-access* (§14.5.7). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the `this` for the instance of the containing type as a constructor argument for the nested type. [*Example:* The following example

```
using System;
class C
{
   int i = 123;
   public void F() {
      Nested n = new Nested(this);
      n.G();
   }

   public class Nested {
      C this_c;
      public Nested(C c) {
         this_c = c;
      }
      public void G() {
         Console.WriteLine(this_c.i);
      }
   }
}

class Test {
   static void Main() {
      C c = new C();
      c.F();
   }
}
```

shows this technique. An instance of `C` creates an instance of `Nested`, and passes its own `this` to `Nested`'s constructor in order to provide subsequent access to `C`'s instance members. *end example*]

17.2.6.5 Access to private and protected members of the containing type

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

```
using System;
class C
{
   private static void F() {
      Console.WriteLine("C.F");
   }
   public class Nested
   {
      public static void G() {
         F();
      }
   }
}

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

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

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

```
using System;
class Base
{
   protected void F() {
      Console.WriteLine("Base.F");
   }
}
```

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

the nested class `Derived.Nested` accesses the protected method `F` defined in `Derived`'s base class, `Base`, by calling through an instance of `Derived`. *end example*]

17.2.7 Reserved member names

To facilitate the underlying C# runtime implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type (§17.2.7.1, §17.2.7.2, §17.2.7.3). It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying runtime implementation does not make use of these reservations.

The reserved names do not introduce declarations, thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance (§17.2.1), and can be hidden with the new modifier (§17.2.2).

[*Note:* The reservation of these names serves three purposes:

1. To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
2. To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
3. To help ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

end note]

The declaration of a destructor (§17.12) also causes a signature to be reserved (§17.2.7.4).

17.2.7.1 Member Names Reserved for Properties

For a property `P` (§17.6) of type `T`, the following signatures are reserved:

```
T get_P();
void set_P(T value);
```

Both signatures are reserved, even if the property is read-only or write-only.
[*Example:* In the example

```
using System;
class A {
   public int P {
      get { return 123; }
   }
}
```

```
class B: A {
   new public int get_P() {
      return 456;
   }
   new public void set_P(int value) {
   }
}
class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      Console.WriteLine(a.P);
      Console.WriteLine(b.P);
      Console.WriteLine(b.get_P());
   }
}
```

a class `A` defines a read-only property `P`, thus reserving signatures for `get_P` and `set_P` methods. A class `B` derives from `A` and hides both of these reserved signatures. The example produces the output:

```
123
123
456
```

end example]

17.2.7.2 Member Names Reserved for Events

For an event `E` (§17.7) of delegate type `T`, the following signatures are reserved:

```
void add_E(T handler);
void remove_E(T handler);
```

17.2.7.3 Member Names Reserved for Indexers

For an indexer (§17.8) 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.

17.2.7.4 Member Names Reserved for Destructors

For a class containing a destructor (§17.12), the following signature is reserved:

```
void Finalize();
```

17.3 Constants

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

constant-declaration:
  attributes$_{opt}$ *constant-modifiers*$_{opt}$ `const` *type* *constant-declarators* `;`

constant-modifiers:
  constant-modifier
  constant-modifiers *constant-modifier*

constant-modifier:
  `new`
  `public`
  `protected`
  `internal`
  `private`

constant-declarators:
constant-declarator
constant-declarators `,` *constant-declarator*

constant-declarator:
identifier = *constant-expression*

A *constant-declaration* may include a set of *attributes* (§24), a `new` modifier (§17.2.2), and a valid combination of the four access modifiers (§17.2.3). The attributes and modifiers apply to all of the members declared by the *constant-declaration*. Even though constants are considered static members, a *constant-declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant-declarator*s, each of which introduces a new member. A *constant-declarator* consists of an *identifier* that names the member, followed by an "=" token, followed by a *constant-expression* (§14.15) 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 (§13.1).

The *type* of a constant must be at least as accessible as the constant itself (§10.5.4).

The value of a constant is obtained in an expression using a *simple-name* (§14.5.2) or a *member-access* (§14.5.4).

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

[*Note:* As described in §14.15, a *constant-expression* is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a *reference-type* other than `string` is to apply the `new` operator, and since the `new` operator is not permitted in a *constant-expression*, the only possible value for constants of *reference-types* other than `string` is `null`. *end note*]

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile-time by a *constant-expression*, a `readonly` field (§17.4.2) may be used instead. [*Note:* The versioning semantics of `const` and `readonly` differ (§17.4.2.2). *end-note*]

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. [*Example:* 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;
}
```

end example]

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. [*Example:* In the 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`. *end example*] Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. [*Example:* Referring to the example above, if `A` and `B` were declared in separate programs, it would be possible for `A.X` to depend on `B.Z`, but `B.Z` could then not simultaneously depend on `A.Y`. *end example*]

17.4 Fields

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

field-declaration:
    attributes$_{opt}$ *field-modifiers$_{opt}$* *type* *variable-declarators* `;`

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

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

variable-declarators:
    variable-declarator
    variable-declarators `,` *variable-declarator*

variable-declarator:
    identifier
    identifier `=` *variable-initializer*

variable-initializer:
    expression
    array-initializer

A *field-declaration* may include a set of *attributes* (§24), a `new` modifier (§17.2.2), a valid combination of the four access modifiers (§17.2.3), and a `static` modifier (§17.4.1). In addition, a *field-declaration* may include a `readonly` modifier (§17.4.2) or a `volatile` modifier (§17.4.3), but not both The attributes and modifiers apply to all of the members declared by the *field-declaration*. It is an error for the same modifier to appear multiple times in a *field declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

The *type* of a *field-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable-declarator*s, each of which introduces a new member. A *variable-declarator* consists of an *identifier* that names that member, optionally followed by an "=" token and a *variable-initializer* (§17.4.5) that gives the initial value of that member.

The *type* of a field must be at least as accessible as the field itself (§10.5.4).

The value of a field is obtained in an expression using a *simple-name* (§14.5.2) or a *member-access* (§14.5.4). The value of a non-readonly field is modified using an *assignment* (§14.13). The value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators (§14.5.9) and prefix increment and decrement operators (§14.6.5).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. [*Example:* 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;
}
```

end example]

17.4.1 Static and instance fields

When a field declaration includes a `static` modifier, the fields introduced by the declaration are ***static fields***. When no `static` modifier is present, the fields introduced by the declaration are ***instance fields***. Static fields and instance fields are two of the several kinds of variables (§12) 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 identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field for the associated application domain.

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* (§14.5.4) of the form `E.M`, if `M` is a static field, `E` must denote a type that has a field `M`, and if `M` is an instance field, `E` must denote an instance of a type that has a field `M`.

The differences between static and instance members are discussed further in §17.2.5.

17.4.2 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.

17.4.2.1 Using static readonly fields for constants

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-time. [*Example:* In the 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. *end example*]

17.4.2.2 Versioning of constants and static readonly fields

Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time. [*Example:* Consider an application that consists of two separate programs:

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

namespace Program2
{
   class Test
   {
      static void Main() {
         Console.WriteLine(Program1.Utils.X);
      }
   }
}
```

The `Program1` and `Program2` namespaces denote two programs that are compiled separately. Because `Program1.Utils.X` is declared as a static readonly field, the value output by the `Console.WriteLine` statement is not known at compile-time, but rather is obtained at run-time. Thus, if the value of `X` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if `Program2` isn't recompiled. However, had `X` been a constant, the value of `X` would have been obtained at the time `Program2` was compiled, and would remain unaffected by changes in `Program1` until `Program2` is recompiled. *end example*]

17.4.3 Volatile fields

When a *field-declaration* includes a `volatile` modifier, the fields introduced by that declaration are ***volatile fields***. For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded programs that access fields without synchronization such as that provided by the *lock-statement* (§15.12). These optimizations can be performed by the compiler, by the runtime system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a ***volatile read***. A volatile read has “acquire semantics”; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a ***volatile write***. A volatile write has “release semantics”; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of the following:

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

[*Example:* The example

```
using System;
using System.Threading;
class Test
{
   public static int result;
   public static volatile bool finished;
   static void Thread2() {
      result = 143;
      finished = true;
   }

   static void Main() {
      finished = false;
      // Run Thread2() in a new thread
      new Thread(new ThreadStart(Thread2)).Start();
      // Wait for Thread2 to signal that it has a result by setting
      // finished to true.
      for (;;) {
         if (finished) {
            Console.WriteLine("result = {0}", result);
            return;
         }
      }
   }
}
```

produces the output:

```
result = 143
```

In this example, the method `Main` starts a new thread that runs the method `Thread2`. This method stores a value into a non-volatile field called `result`, then stores `true` in the volatile field `finished`. The main thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `result` 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. *end example*]

17.4.4 Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (§12.2) 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”. [*Example:* The example

```
using System;
class Test
{
   static bool b;
   int i;

   static void Main() {
      Test t = new Test();
      Console.WriteLine("b = {0}, i = {1}", b, t.i);
   }
}
```

produces the output

```
b = False, i = 0
```

because `b` and `i` are both automatically initialized to default values. *end example*]

17.4.5 Variable initializers

Field declarations may include *variable-initializer*s. For static fields, variable initializers correspond to assignment statements that are executed during class initialization. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

[*Example:* The example

```
using System;
class Test
{
   static double x = Math.Sqrt(2.0);
   int i = 100;
   string s = "Hello";

   static void Main() {
      Test a = new Test();
      Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
   }
}
```

produces the output

```
x = 1.4142135623731, i = 100, s = Hello
```

because an assignment to `x` occurs when static field initializers execute and assignments to `i` and `s` occur when the instance field initializers execute. *end example*]

The default value initialization described in §17.4.3 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. [*Example:* However, this is strongly discouraged as a matter of style. The example

```
using System;
class Test
{
   static int a = b + 1;
   static int b = a + 1;

   static void Main() {
      Console.WriteLine("a = {0}, b = {1}", a, b);
   }
}
```

exhibits this behavior. Despite the circular definitions of `a` and `b`, the program is valid. It results in the output

```
a = 1, b = 2
```

because the static fields a and b are initialized to 0 (the default value for int) before their initializers are executed. When the initializer for a runs, the value of b is zero, and so a is initialized to 1. When the initializer for b runs, the value of a is already 1, and so b is initialized to 2. *end example*]

17.4.5.1 Static field initialization

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor (§17.11) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. [*Example*: The example

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine("{0} {1}", B.Y, A.X);
   }
   public static int f(string s) {
      Console.WriteLine(s);
      return 1;
   }
}
class A
{
   public static int X = Test.f("Init A");
}
class B
{
   public static int Y = Test.f("Init B");
}
```

might produce either the output:

```
Init A
Init B
1 1
```

or the output:

```
Init B
Init A
1 1
```

because the execution of X's initializer and Y's initializer could occur in either order; they are only constrained to occur before the references to those fields. However, in the example:

```
using System;
class Test {
   static void Main() {
      Console.WriteLine("{0} {1}", B.Y, A.X);
   }
   public static int f(string s) {
      Console.WriteLine(s);
      return 1;
   }
}
class A
{
   static A() {}
   public static int X = Test.f("Init A");
}
```

```
class B
{
   static B() {}
   public static int Y = Test.f("Init B");
}
```

the output must be:

```
Init B
Init A
1 1
```

because the rules for when static constructors execute provide that `B`'s static constructor (and hence `B`'s static field initializers) must run before `A`'s static constructor and field initializers. *end example*]

17.4.5.2 Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (§17.10.2) 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 §17.10.

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*. [*Example:* 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. *end example*]

17.5 Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class. Methods are declared using *method-declaration*s:

method-declaration:
  method-header method-body

method-header:
  attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type member-name* `(` *formal-parameter-list*$_{opt}$ `)`

method-modifiers:
  method-modifier
  method-modifiers method-modifier

method-modifier:
  `new`
  `public`
  `protected`
  `internal`
  `private`
  `static`
  `virtual`
  `sealed`
  `override`
  `abstract`
  `extern`

return-type:
type
`void`

member-name:
identifier
interface-type `.` *identifier*

method-body:
block
`;`

A *method-declaration* may include a set of *attributes* (§24) and a valid combination of the four access modifiers (§17.2.3), the `new` (§17.2.2), `static` (§17.5.2), `virtual` (§17.5.3), `override` (§17.5.4), `sealed` (§17.5.5), `abstract` (§17.5.6), and `extern` (§17.5.7) 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 (§17.2.3).
- 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`, 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.

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.

The *member-name* specifies the name of the method. Unless the method is an explicit interface member implementation (§20.4.1), 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 *formal-parameter-list* specifies the parameters of the method (§17.5.1).

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 (§10.5.4).

For `abstract` and `extern` methods, the *method-body* consists simply of a semicolon. For all other methods, the *method-body* consists of a *block*, which specifies the statements to execute when the method is invoked.

The name and the formal parameter list of a method define the signature (§10.6) of the method. Specifically, the signature of a method consists of its name and the number, modifiers, and types of its formal parameters. The return type is not part of a method's signature, nor are the names of the formal parameters.

The 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.

17.5.1 Method parameters

The parameters of a method, if any, are declared by the method's *formal-parameter-list*.

formal-parameter-list:
  fixed-parameters
  fixed-parameters `,` *parameter-array*
  parameter-array

fixed-parameters:
  fixed-parameter
  fixed-parameters `,` *fixed-parameter*

fixed-parameter:
  $attributes_{opt}$ $parameter\text{-}modifier_{opt}$ *type* *identifier*

parameter-modifier:
  `ref`
  `out`

parameter-array:
  $attributes_{opt}$ `params` *array-type* *identifier*

The formal parameter list consists of one or more comma-separated parameters of which only the last may be a *parameter-array*.

A *fixed-parameter* consists of an optional set of *attributes* (§24), an optional `ref` or `out` modifier, a *type*, and an *identifier*. Each *fixed-parameter* declares a parameter of the given type with the given name.

A *parameter-array* consists of an optional set of *attributes* (§24), a `params` modifier, an *array-type*, and an *identifier*. A parameter array declares a single parameter of the given array type with the given name. The *array-type* of a parameter array must be a single-dimensional array type (§19.1). In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. Parameter arrays are described further in §17.5.1.4.

A method declaration creates a separate declaration space for parameters and local variables. Names are introduced into this declaration space by the formal parameter list of the method and by local variable declarations in the *block* of the method. All names in the declaration space of a method must be unique. Thus, it is a compile-time error for a parameter or local variable to have the same name as another parameter or local variable.

A method invocation (§14.5.5.1) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple-name* expressions (§14.5.2).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the `ref` modifier.
- Output parameters, which are declared with the `out` modifier.
- Parameter arrays, which are declared with the `params` modifier.

[*Note:* As described in §10.6, the `ref` and `out` modifiers are part of a method's signature, but the `params` modifier is not. *end note*]

17.5.1.1 Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible (§13.1) to the formal parameter type.

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.

17.5.1.2 Reference parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a *variable-reference* (§12.3.3) 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.

[*Example:* The example

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

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

produces the output

```
i = 2, j = 1
```

For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus, the invocation has the effect of swapping the values of `i` and `j`. *end example*]

In a method that takes reference parameters, it is possible for multiple names to represent the same storage location. [*Example:* In the 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`. *end example*]

17.5.1.3 Output parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable-reference* (§12.3.3) 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.

Output parameters are typically used in methods that produce multiple return values. [*Example:* For example:

```
using System;
class Test
{
   static void SplitPath(string path, out string dir, out string name) {
      int i = path.Length;
      while (i > 0) {
         char ch = path[i – 1];
         if (ch == '\\' || ch == '/' || ch == ':') break;
         i--;
      }
      dir = path.Substring(0, i);
      name = path.Substring(i);
   }

   static void Main() {
      string dir, name;
      SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
      Console.WriteLine(dir);
      Console.WriteLine(name);
   }
}
```

The example produces the output:

```
c:\Windows\System\
hello.txt
```

Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they are considered definitely assigned following the call. *end example*]

17.5.1.4 Parameter arrays

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. [*Example:* For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. *end example*] It is not possible to combine the `params` modifier with the modifiers `ref` and `out`.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression of a type that is implicitly convertible (§13.1) 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 of a type that is implicitly convertible (§13.1) 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 (§17.5.1.1) of the same type.

[*Example:* The example

```
using System;
class Test
{
   static void F(params int[] args) {
      Console.Write("Array contains {0} elements:", args.Length);
      foreach (int i in args)
         Console.Write(" {0}", i);
      Console.WriteLine();
   }

   static void Main() {
      int[] arr = {1, 2, 3};
      F(arr);
      F(10, 20, 30, 40);
      F();
   }
}
```

produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `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[] {});
```

end example]

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form (§14.4.2.1). The expanded form of a method is available only if the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type.

[*Example:* The example

```
using System;
class Test
{
   static void F(params object[] a) {
      Console.WriteLine("F(object[])");
   }

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

   static void F(object a0, object a1) {
      Console.WriteLine("F(object,object)");
   }

   static void Main() {
      F();
      F(1);
      F(1, 2);
      F(1, 2, 3);
      F(1, 2, 3, 4);
   }
}
```

produces the output

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked. *end example*]

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the 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.

[*Example:* The example

```
using System;
class Test
{
   static void F(params object[] args) {
      foreach (object o in args) {
         Console.Write(o.GetType().FullName);
         Console.Write(" ");
      }
      Console.WriteLine();
   }

   static void Main() {
      object[] a = {1, "Hello", 123.456};
      object o = a;
      F(a);
      F((object)a);
      F(o);
      F((object[])o);
   }
}
```

produces the output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of F, the normal form of `F` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects the normal form of F, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `F` is applicable, so it is selected by overload resolution. As a result, a one-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`). *end example*]

17.5.2 Static and instance methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this` (§14.5.7).

When a method is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static method, `E` must denote a type that has a method `M`, and if `M` is an instance method, `E` must denote an instance of a type that has a method `M`.

The differences between static and instance members are discussed further in §17.2.5.

17.5.3 Virtual methods

When an instance method declaration includes a `virtual` modifier, that method is said to be a ***virtual method***. When no `virtual` modifier is present, the method is said to be a ***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 (§17.5.4).

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 §14.5.5.1.
- 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` is the same as that of the direct base class of `R`.

[*Example:* The following example illustrates the differences between virtual and non-virtual methods:

```
using System;
class A
{
   public void F() { Console.WriteLine("A.F"); }
   public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
   new public void F() { Console.WriteLine("B.F"); }
   public override void G() { Console.WriteLine("B.G"); }
}
```

```
class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      a.F();
      b.F();
      a.G();
      b.G();
   }
}
```

In the example, A introduces a non-virtual method F and a virtual method G. The class B introduces a *new* non-virtual method F, thus *hiding* the inherited F, and also *overrides* the inherited method G. The example produces the output:

```
A.F
B.F
B.G
B.G
```

Notice that the statement a.G() invokes B.G, not A.G. This is because the run-time type of the instance (which is B), not the compile-time type of the instance (which is A), determines the actual method implementation to invoke. *end example*]

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. [*Example:* In the example

```
using System;
class A
{
   public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
   public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
   new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
   public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
   static void Main() {
      D d = new D();
      A a = d;
      B b = d;
      C c = d;
      a.F();
      b.F();
      c.F();
      d.F();
   }
}
```

the C and D classes contain two virtual methods with the same signature: The one introduced by A and the one introduced by C. The method introduced by C hides the method inherited from A. Thus, the override declaration in D overrides the method introduced by C, and it is not possible for D to override the method introduced by A. The example produces the output:

```
B.F
B.F
D.F
D.F
```

Note that it is possible to invoke the hidden virtual method by accessing an instance of `D` through a less derived type in which the method is not hidden. *end example*]

17.5.4 Override methods

When an instance method declaration includes an `override` modifier, the method is said to be an ***override method***. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration *introduces* a new method, an override method declaration *specializes* an existing inherited virtual method by providing a new implementation of that method.

The method overridden by an `override` declaration is known as the ***overridden base method***. For an override method `M` declared in a class C, the overridden base method is determined by examining each base class of C, starting with the direct base class of `C` and continuing with each successive direct base class, until an accessible method with the same signature as `M` is located. 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.
- 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 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.

An override declaration can access the overridden base method using a *base-access* (§14.5.8). [*Example:* In the example

```
class A
{
   int x;
   public virtual void PrintFields() {
      Console.WriteLine("x = {0}", x);
   }
}
class B: A
{
   int y;
   public override void PrintFields() {
      base.PrintFields();
      Console.WriteLine("y = {0}", y);
   }
}
```

the `base.PrintFields()` invocation in `B` invokes the `PrintFields` method declared in `A`. A *base-access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `B` been written `((A)this).PrintFields()`, it would recursively invoke the `PrintFields` method declared in B, not the one declared in A, since `PrintFields` is virtual and the run-time type of `((A)this)` is `B`. *end example*]

Only by including an `override` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. [*Example:* In the 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. *end example*]

[*Example:* In the example

```
class A
{
   public virtual void F() {}
}
class B: A
{
   new private void F() {}       // Hides A.F within B
}
class C: B
{
   public override void F() {}   // Ok, overrides A.F
}
```

the F method in B hides the virtual F method inherited from A. Since the new F in B has private access, its scope only includes the class body of B and does not extend to C. Therefore, the declaration of F in C is permitted to override the F inherited from A. *end example*]

17.5.5 Sealed methods

When an instance method declaration includes a sealed modifier, that method is said to be a ***sealed method***. A sealed method overrides an inherited virtual method with the same signature. An override method can also be marked with the sealed modifier. Use of this modifier prevents a derived class from further overriding the method.

[*Example:* The example

```
using System;
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
   public virtual void G() {
      Console.WriteLine("A.G");
   }
}
class B: A
{
   sealed override public void F() {
      Console.WriteLine("B.F");
   }
   override public void G() {
      Console.WriteLine("B.G");
   }
}
```

```
class C: B
{
   override public void G() {
      Console.WriteLine("C.G");
   }
}
```

the class `B` provides two override methods: an `F` method that has the `sealed` modifier and a `G` method that does not. `B`'s use of the sealed `modifier` prevents `C` from further overriding `F`. *end example*]

17.5.6 Abstract methods

When an instance method declaration includes an `abstract` modifier, that method is said to be an ***abstract method***. Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, 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 (§17.1.1.1).

[*Example:* 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. *end example*]

It is a compile-time error for a *base-access* (§14.5.8) to reference an abstract method. [*Example:* 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. *end example*]

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. [*Example:* In the example

```
using System;
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
}
abstract class B: A
{
   public abstract override void F();
}
class C: B
{
   public override void F() {
      Console.WriteLine("C.F");
   }
}
```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides that abstract method to provide its own implementation. *end example*]

17.5.7 External methods

When a method declaration includes an `extern` modifier, the method is said to be an ***external method***. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method-body* of an external method simply consists of a semicolon.

The mechanism by which linkage to an external method is achieved, is implementation-defined.

[*Example:* The following example demonstrates the use of the `extern` modifier in combination with a `DllImport` attribute that specifies the name of the external library in which the method is implemented:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;
class Path
{
   [DllImport("kernel32", SetLastError=true)]
   static extern bool CreateDirectory(string name, SecurityAttribute sa);

   [DllImport("kernel32", SetLastError=true)]
   static extern bool RemoveDirectory(string name);

   [DllImport("kernel32", SetLastError=true)]
   static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

   [DllImport("kernel32", SetLastError=true)]
   static extern bool SetCurrentDirectory(string name);
}
```

end example]

17.5.8 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 (§15.2) that contains the statements to execute when that method is invoked.

When the return type of a method is `void`, `return` statements (§15.9.4) 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 caller.

When the return type of a method is not `void`, each `return` statement in that method body must specify an expression of a type that is implicitly convertible to the return 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.

[*Example:* 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. *end example*]

17.5.9 Method overloading

The method overload resolution rules are described in §14.4.2.

17.6 Properties

A ***property*** is a member that provides access to an attribute of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes; furthermore, they permit such attributes to be computed.

Properties are declared using *property-declaration*s:

property-declaration:
  attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* `{` *accessor-declarations* `}`

property-modifiers:
  property-modifier
  property-modifiers *property-modifier*

property-modifier:
```
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern
```

member-name:
  identifier
  interface-type `.` *identifier*

A *property-declaration* may include a set of *attributes* (§24) and a valid combination of the four access modifiers (§17.2.3), the `new` (§17.2.2), `static` (§17.6.1), `virtual` (§17.5.3, §17.6.3), `override` (§17.5.4, §17.6.3), `sealed` (§17.5.5), `abstract` (§17.5.6, §17.6.3), and `extern` modifiers.

Property declarations are subject to the same rules as method declarations (§17.5) 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 (§20.4.1), 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 (§10.5.4).

The *accessor-declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (§17.6.2) 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.

17.6.1 Static and instance properties

When a property declaration includes a `static` modifier, the property is said to be a ***static property***. When no `static` modifier is present, the property is said to be an ***instance property***.

A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as `this` (§14.5.7) in the accessors of that property.

When a property is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static property, `E` must denote a type that has a property `M`, and if `M` is an instance property, E must denote an instance having a property `M`.

The differences between static and instance members are discussed further in §17.2.5.

17.6.2 Accessors

The *accessor-declarations* of a property specify the executable statements associated with reading and writing that property.

> *accessor-declarations:*
> *get-accessor-declaration set-accessor-declaration$_{opt}$*
> *set-accessor-declaration get-accessor-declaration$_{opt}$*
>
> *get-accessor-declaration:*
> *attributes$_{opt}$* `get` *accessor-body*
>
> *set-accessor-declaration:*
> *attributes$_{opt}$* `set` *accessor-body*
>
> *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 *accessor-body*. For `abstract` and `extern` properties, the *accessor-body* for each accessor specified is simply a semicolon. For the accessors of any 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 (§14.1.1). The body of a `get` accessor must conform to the rules for value-returning methods described in §17.5.8. 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 (§14.13), or as the operand of ++ or `--` (§14.5.9, 14.6.5), 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 (§14.13.1). The body of a `set` accessor must conform to the rules for `void` methods described in §17.5.8. 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 declaration in a `set` accessor to have that name.

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

- A property that includes both a `get` accessor and a `set` accessor is said to be a ***read-write*** property.
- A property that has only a `get` accessor is said to be a ***read-only*** property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a ***write-only*** property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression. [*Note:* The pre- and postfix ++ and `--` operators cannot be applied to write-only properties, since these operators read the old value of their operand before they write the new one. *end note*]

[*Example:* In the 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. *end example*]

The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. [*Note:* As such, it is not possible for the two accessors of a read-write property to have different accessibility. *end note*] [*Example:* 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. *end example*]

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. [*Example:* In the 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. *end example*]

Unlike public fields, properties provide a separation between an object's internal state and its public interface. [*Example:* 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. *end example*]

[*Note:* Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment 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. *end note*]

[*Example:* Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for `get` accessors to have observable side-effects. In the example

```
class Counter
{
   private int next;

   public int Next {
      get { return next++; }
   }
}
```

the value of the `Next` property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side effect, and the property should be implemented as a method instead. *end example*]

[*Note:* 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. *end note*]

Properties can be used to delay initialization of a resource until the moment it is first referenced. [*Example:* 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. *end example*]

17.6.3 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 (§17.1.1.1). 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 §17.5.3, §17.5.4, §17.5.5, and §17.5.6 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.

[*Example:* 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. *end example*]

17.7 Events

An ***event*** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying ***event handlers***.

Events are declared using *event-declaration*s:

event-declaration:
  attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *variable-declarators* `;`
  attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *member-name* `{` *event-accessor-declarations* `}`

event-modifiers:
  event-modifier
  event-modifiers *event-modifier*

event-modifier:
  `new`
  `public`
  `protected`
  `internal`
  `private`
  `static`
  `virtual`
  `sealed`
  `override`
  `abstract`
  `extern`

event-accessor-declarations:
  add-accessor-declaration *remove-accessor-declaration*
  remove-accessor-declaration *add-accessor-declaration*

add-accessor-declaration:
  attributes$_{opt}$ `add` *block*

remove-accessor-declaration:
  attributes$_{opt}$ `remove` *block*

An *event-declaration* may include a set of *attributes* (§24) and a valid combination of the four access modifiers (§17.2.3), the `new` (§17.2.2), `static` (§17.5.2, §17.7.3), `virtual` (§17.5.3, §17.7.4), `override` (§17.5.4, §17.7.4), `sealed` (§17.5.5), `abstract` (§17.5.6, §17.7.4), and `extern` modifiers.

Event declarations are subject to the same rules as method declarations (§17.5) with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate-type* (§11.2), and that *delegate-type* must be at least as accessible as the event itself (§10.5.4).

An event declaration may include *event-accessor-declaration*s. However, if it does not, for non-extern, non-abstract events, the compiler shall supply them automatically (§17.7.1); for extern events, the accessors are provided externally.

An event declaration that omits *event-accessor-declaration*s defines one or more events—one for each of the *variable-declarator*s. The attributes and modifiers apply to all of the members declared by such an *event-declaration*.

It is a compile-time error for an *event-declaration* to include both the `abstract` modifier and brace-delimited *event-accessor-declaration*s.

When an event declaration includes an `extern` modifier, the event is said to be an ***external event***. Because an external event declaration provides no actual implementation, it is an error for it to include both the `extern` modifier and *event-accessor-declaration*s.

An event can be used as the left-hand operand of the += and `-=` operators (§14.13.3). 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.

[*Example:* The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
   public event EventHandler Click;
}

public class LoginDialog: Form
{
   Button OkButton;
   Button CancelButton;

   public LoginDialog() {
      OkButton = new Button(…);
      OkButton.Click += new EventHandler(OkButtonClick);
      CancelButton = new Button(…);
      CancelButton.Click += new EventHandler(CancelButtonClick);
   }

   void OkButtonClick(object sender, EventArgs e) {
      // Handle OkButton.Click event
   }
```

```
            void CancelButtonClick(object sender, EventArgs e) {
               // Handle CancelButton.Click event
            }
         }
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events. *end example*]

17.7.1 Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly include *event-accessor-declaration*s. Such an event can be used in any context that permits a field. The field contains a delegate (§22), which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `null`.

[*Example:* In the 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. *end example*]

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. In order to be thread-safe, the addition or removal operations are done while holding the lock (§15.12) on the containing object for an instance event, or the type object (§14.5.11) for a static event.

[*Note:* Thus, an instance event declaration of the form:

```
class X {
   public event D Ev;
}
```

could be compiled to something equivalent to:

```
class X {
   private D __Ev;  // field to hold the delegate
   public event D Ev {
      add {
         lock(this) { __Ev = __Ev + value; }
      }
```

```
            remove {
               lock(this) { __Ev = __Ev - value; }
            }
         }
      }
```

Within the class `X`, references to `Ev` are compiled to reference the hidden field `__Ev` instead. The name "`__Ev`" is arbitrary; the hidden field could have any name or no name at all.

Similarly, a static event declaration of the form:

```
      class X {
         public static event D Ev;
      }
```

could be compiled to something equivalent to:

```
      class X {
         private static D __Ev;  // field to hold the delegate

         public static event D Ev {
            add {
               lock(typeof(X)) { __Ev = __Ev + value; }
            }

            remove {
               lock(typeof(X)) { __Ev = __Ev - value; }
            }
         }
      }
```

end note]

17.7.2 Event accessors

[*Note:* Event declarations typically omit *event-accessor-declaration*s, as in the `Button` example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include *event-accessor-declaration*s and use a private mechanism for storing the list of event handlers. Similarly, in cases where the handling of an event requires access to external resources, event accessors may be used to manage these resources. *end note*]

The *event-accessor-declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add-accessor-declaration* and a *remove-accessor-declaration*. Each accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an *add-accessor-declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove-accessor-declaration* specifies the statements to execute when an event handler is removed.

Each *add-accessor-declaration* and *remove-accessor-declaration* corresponds to a method with a single value parameter of the event type, and a `void` return type. The implicit parameter of an event accessor is named `value`. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is += then the add accessor is used, and if the assignment operator is –= then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add-accessor-declaration* or a *remove-accessor-declaration* must conform to the rules for `void` methods described in §17.5.8. 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 declared in an event accessor to have that name.

[*Example:* 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. *end example*]

17.7.3 Static and instance events

When an event declaration includes a `static` modifier, the event is said to be a ***static event***. When no `static` modifier is present, the event is said to be an ***instance event***.

A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as `this` (§14.5.7) in the accessors of that event.

When an event is referenced in a *member-access* (§14.5.4) of the form `E.M`, if `M` is a static event, `E` must denote a type, and if `M` is an instance event, E must denote an instance.

The differences between static and instance members are discussed further in §17.2.5.

17.7.4 Virtual, sealed, override, and abstract accessors

A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier applies to both accessors of an event.

An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an accessor for an abstract event declaration provides no actual implementation, its *accessor-body* simply consists of a semicolon.

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 (§17.1.1.1).

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 §17.5.3, §17.5.4, §17.5.5, and §17.5.6 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.

17.8 Indexers

An ***indexer*** is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer-declaration*s:

indexer-declaration:
  attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* `{` *accessor-declarations* `}`

indexer-modifiers:
  indexer-modifier
  indexer-modifiers *indexer-modifier*

indexer-modifier:
  `new`
  `public`
  `protected`
  `internal`
  `private`
  `virtual`
  `sealed`
  `override`
  `abstract`
  `extern`

indexer-declarator:
  type `this` `[` *formal-parameter-list* `]`
  type *interface-type* `.` `this` `[` *formal-parameter-list* `]`

An *indexer-declaration* may include a set of *attributes* (§24) and a valid combination of the four access modifiers (§17.2.3), the `new` (§17.2.2), `virtual` (§17.5.3), `override` (§17.5.4), `sealed` (§17.5.5), `abstract` (§17.5.6), and `extern` (§17.5.7) modifiers.

Indexer declarations are subject to the same rules as method declarations (§17.5) 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 (§17.5.1), 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 (§10.5.4).

The *accessor-declarations* (§17.6.2), 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 (§10.6) 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* (§14.5.2) or a *member-access* (§14.5.4), whereas an indexer element is accessed through an *element-access* (§14.5.6.2).
- 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 §17.6.2 and §17.6.3 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.

[*Example:* The example below declares a `BitArray` class that implements an indexer for accessing the individual bits in the bit array.

```
using System;
class BitArray
{
   int[] bits;
   int length;

   public BitArray(int length) {
      if (length < 0) throw new ArgumentException();
      bits = new int[((length - 1) >> 5) + 1];
      this.length = length;
   }

   public int Length {
      get { return length; }
   }

   public bool this[int index] {
      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);
      Console.WriteLine("Found {0} primes between 1 and {1}", count,
max);
   }
}
```

Note that the syntax for accessing elements of the `BitArray` is precisely the same as for a `bool[]`. *end example*]

[*Example:* The following example shows a 26×10 grid class that has an indexer with two parameters. The first parameter is required to be an upper- or lowercase letter in the range A–Z, and the second is required to be an integer in the range 0–9.

```
using System;
class Grid
{
   const int NumRows = 26;
   const int NumCols = 10;
   int[,] cells = new int[NumRows, NumCols];

   public int this[char c, int colm]
   {
      get {
         c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z') {
            throw new ArgumentException();
         }
         if (colm < 0 || colm >= NumCols) {
            throw new IndexOutOfRangeException();
         }
         return cells[c - 'A', colm];
      }
      set {
         c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z') {
            throw new ArgumentException();
         }
         if (colm < 0 || colm >= NumCols) {
            throw new IndexOutOfRangeException();
         }
         cells[c - 'A', colm] = value;
      }
   }
}
```

end example]

17.8.1 Indexer overloading

The indexer overload resolution rules are described in §14.4.2.

17.9 Operators

An ***operator*** is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using *operator-declaration*s:

operator-declaration:
    attributes$_{opt}$ operator-modifiers operator-declarator operator-body

operator-modifiers:
    operator-modifier
    operator-modifiers operator-modifier

operator-modifier:
    `public`
    `static`
    `extern`

operator-declarator:
    unary-operator-declarator
    binary-operator-declarator
    conversion-operator-declarator

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: one of
`+ - * / % & | ^ << >> == != > < >= <=`

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 (§17.9.1), binary operators (§17.9.2), and conversion operators (§17.9.3).

When an operator declaration includes an `extern` modifier, the operator is said to be an ***external operator***. Because an external operator provides no actual implementation, its *operator-body* consists of a semi-colon. For all other operators, the *operator-body* consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* of an operator must conform to the rules for value-returning methods described in §17.5.8.

The following rules apply to all operator declarations:

- An operator declaration must include both a `public` and a `static` modifier.
- The parameter(s) of an operator must be value parameters. It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator (§17.9.1, §17.9.2, §17.9.3) must differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration must be at least as accessible as the operator itself (§10.5.4).
- It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the `new` modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in §14.2.

Additional information on conversion operators can be found in §13.4.

17.9.1 Unary operators

The following rules apply to unary operator declarations, where `T` denotes the class or struct type that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type `T` and can return any type.
- A unary `++` or `--` operator must take a single parameter of type `T` and must return type `T`.

- A unary `true` or `false` operator must take a single parameter of type `T` and must return type `bool`.

The signature of a unary operator consists of the operator token (+, -, !, ~, ++, --, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The `true` and `false` unary operators require pair-wise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in §14.16.

[*Example:* The following example shows an implementation and subsequent usage of `operator++` for an integer vector class:

```
public class IntVector
{
   public int Length { … }          // read-only property
   public int this[int index] { … } // read-write indexer
   public IntVector(int vectorLength) { … }
   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 4x0
      IntVector iv2;

      iv2 = iv1++;   // iv2 contains 4x0, iv1 contains 4x1
      iv2 = ++iv1;   // iv2 contains 4x2, iv1 contains 4x2
}
```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators(§14.5.9), and the prefix increment and decrement operators (§14.6.5). Unlike in C++, this method need not, and, in fact, must not, modify the value of its operand directly. *end example*]

17.9.2 Binary operators

A binary operator must take two parameters, at least one of which must have the class or struct type in which the operator is declared. A binary operator can return any type.

The signature of a binary operator consists of the operator token (+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

- `operator` == and `operator` !=
- `operator` > and `operator` <
- `operator` >= and `operator` <=

17.9.3 Conversion operators

A conversion operator declaration introduces a ***user-defined conversion*** (§13.4), which augments the pre-defined implicit and explicit conversions.

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 §13.1.

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 §13.2.

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. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` provided all of the following are true:

- `S` and `T` are different types.
- Either `S` or `T` is the class or struct type in which the operator declaration takes place.
- Neither `S` nor `T` is `object` or an *interface-type*.
- `T` is not a base class of S, and `S` is not a base class of 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. [*Example:* 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`. *end example*]

It is not possible to 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.

User-defined conversions are not allowed to convert from or to *interface-type*s. In particular, this restriction ensures that no user-defined transformations occur when converting to an *interface-type*, and that a conversion to an *interface-type* succeeds only if the object being converted actually implements the specified *interface-type*.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The `implicit` or `explicit` classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an `implicit` and an `explicit` conversion operator with the same source and target types.

[*Note:* In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion. *end note*]

[*Example:* In the 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`. *end example*]

17.10 Instance constructors

An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor-declaration*s:

constructor-declaration:
    attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator* *constructor-body*

constructor-modifiers:
    constructor-modifier
    constructor-modifiers *constructor-modifier*

constructor-modifier:
    `public`
    `protected`
    `internal`
    `private`
    `extern`

constructor-declarator:
    identifier `(` *formal-parameter-list*$_{opt}$ `)` *constructor-initializer*$_{opt}$

constructor-initializer:
    `: base (` *argument-list*$_{opt}$ `)`
    `: this (` *argument-list*$_{opt}$ `)`

constructor-body:
    block
    `;`

A *constructor-declaration* may include a set of *attributes* (§24), a valid combination of the four access modifiers (§17.2.3), and an `extern` (§17.5.7) 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 (§17.5). The formal parameter list defines the signature (§10.6) of an instance constructor and governs the process whereby overload resolution (§14.4.2) 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 (§10.5.4).

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 §17.10.1.

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 (§17.5.8).

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 (§17.10.4).

Instance constructors are invoked by *object-creation-expression*s (§14.5.10.1) and through *constructor-initializer*s.

17.10.1 Constructor initializers

All instance constructors (except those for class `object`) implicitly include an invocation of another instance constructor immediately before the *constructor-body*. The constructor to implicitly invoke is determined by the *constructor-initializer*:

- An instance constructor initializer of the form `base(`*argument-list*$_{opt}$`)` causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument-list* and the overload resolution rules of §14.4.2. The set of candidate instance constructors consists of all accessible instance constructors declared in the direct base class. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
- An instance constructor initializer of the form `this(`*argument-list*$_{opt}$`)` causes an instance constructor from the class itself to be invoked. The constructor is selected using *argument-list* and the overload resolution rules of §14.4.2. The set of candidate instance constructors consists of all accessible instance constructors declared in the class itself. If that 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. [*Note:* Thus, an instance constructor declaration of the form

```
C(…) {…}
```

is exactly equivalent to

```
C(…): base() {…}
```

end note]

The scope of the parameters given by the *formal-parameter-list* of an instance constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. [*Example:* For example:

```
class A
{
   public A(int x, int y) {}
}
class B: A
{
   public B(int x, int y): base(x + y, x - y) {}
}
```

end example]

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference `this` in an argument expression of the constructor initializer, as it is a compile-time error for an argument expression to reference any instance member through a *simple-name*.

17.10.2 Instance variable initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(…)`, that constructor implicitly performs the initializations specified by the *variable-initializer*s of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor. The variable initializers are executed in the textual order in which they appear in the class declaration.

17.10.3 Constructor execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed *before* the invocation of the base class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before *any* statements that have access to that instance are executed. [*Example:* For example:

```
using System;
class A
{
   public A() {
      PrintFields();
   }

   public virtual void PrintFields() {}
}

class B: A
{
   int x = 1;
   int y;

   public B() {
      y = -1;
   }

   public override void PrintFields() {
      Console.WriteLine("x = {0}, y = {1}", x, y);
   }
}
```

When `new B()` is used to create an instance of B, the following output is produced:

```
x = 1, y = 0
```

The value of `x` is `1` because the variable initializer is executed before the base class instance constructor is invoked. However, the value of `y` is `0` (the default value of an `int`) because the assignment to `y` is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor-body*. [*Example:* 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;
   }
}
```

end example]

17.10.4 Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless instance constructor, a compile-time error occurs. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. [*Note:* Thus, the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where `C` is the name of the class. *end note*]

[*Example:* 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() {}
}
```

end example]

17.10.5 Private constructors

When a class declares only private instance constructors, it is not possible for other classes to derive from that class or to create instances of that class (an exception being classes nested within that class). [*Example:* Private instance constructors are commonly used in classes that contain only static members. 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. *end example*] At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

17.10.6 Optional instance constructor parameters

[*Note:* 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");
```

end note]

17.11 Static constructors

A ***static constructor*** is a member that implements the actions required to initialize a class. Static constructors are declared using *static-constructor-declaration*s:

static-constructor-declaration:
 attributes$_{opt}$ *static-constructor-modifiers* *identifier* `(` `)` *static-constructor-body*

static-constructor-modifiers:
`extern`*opt* `static`
`static extern`*opt*

static-constructor-body:
block
`;`

A *static-constructor-declaration* may include a set of *attributes* (§24) and an `extern` modifier (§17.5.7).

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 (§17.5.8).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a class executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

- An instance of the class is created.
- Any of the static members of the class are referenced.

If a class contains the `Main` method (§10.1) in which execution begins, the static constructor for that class executes before the `Main` method is called. If a class contains any static fields with initializers, those initializers are executed in textual order immediately prior to executing the static constructor.

[*Example*: The example

```
using System;
class Test
{
   static void Main() {
      A.F();
      B.F();
   }
}
class A
{
   static A() {
      Console.WriteLine("Init A");
   }
   public static void F() {
      Console.WriteLine("A.F");
   }
}
class B
{
   static B() {
      Console.WriteLine("Init B");
   }
   public static void F() {
      Console.WriteLine("B.F");
   }
}
```

must produce the output:

```
Init A
A.F
Init B
B.F
```

because the execution of `A`'s static constructor is triggered by the call to `A.F`, and the execution of `B`'s static constructor is triggered by the call to `B.F`. *end example*]

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

[*Example*: The example

```
using System;
class A
{
   public static int X;
   static A() { X = B.Y + 1;}
}

class B
{
   public static int Y = A.X + 1;
   static B() {}
   static void Main() {
      Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
   }
}
```

produces the output

```
X = 1, Y = 2
```

To execute the `Main` method, the system first runs the initializer for `B.Y`, prior to class `B`'s static constructor. `Y`'s initializer causes `A`'s static constructor to be run because the value of `A.X` is referenced. The static constructor of `A` in turn proceeds to compute the value of `X`, and in doing so fetches the default value of `Y`, which is zero. `A.X` is thus initialized to 1. The process of running `A`'s static field initializers and static constructor then completes, returning to the calculation of the initial value of `Y`, the result of which becomes 2. *end example*]

17.12 Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a *destructor-declaration*:

destructor-declaration:
  attributes$_{opt}$ `extern`$_{opt}$ `~` *identifier* `( )` *destructor-body*

destructor-body:
  block
  `;`

A *destructor-declaration* may include a set of *attributes* (§24).

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 (§17.5.8).

Destructors are not inherited. Thus, a class has no destructors other than the one which may be declared in that class.

[*Note:* Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor. *end note*]

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 [*Example:* The output of the example

```
using System;
class A
{
   ~A() {
      Console.WriteLine("A's destructor");
   }
}
class B: A
{
   ~B() {
      Console.WriteLine("B's destructor");
   }
}
class Test
{
   static void Main() {
      B b = new B();
      b = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();
   }
}
```

is

```
B’s destructor
A’s destructor
```

since destructors in an inheritance chain are called in order, from most derived to least derived. *end example*]

Destructors may be implemented by overriding the virtual method `Finalize` on `System.Object`. In any event, C# programs are not permitted to override this method or call it (or overrides of it) directly. [*Example:* For instance, the program

```
class A
{
   override protected void Finalize() {}  // error
   public void F() {
      this.Finalize();     // error
   }
}
```

contains two errors. *end example*]

The compiler behaves as if this method, and overrides of it, does not exist at all. [*Example:* Thus, this program:

```
class A
{
   void Finalize() {}      // permitted
}
```

is valid and the method shown hides `System.Object`'s `Finalize` method. *end example*]

For a discussion of the behavior when an exception is thrown from a destructor, see §23.3.

18. Structs

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

[*Note:* Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference. *end note*]

As described in §11.1.3, 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 (§18.4).

18.1 Struct declarations

A *struct-declaration* is a *type-declaration* (§16.5) that declares a new struct:

> *struct-declaration:*
>   *attributes*$_{opt}$ *struct-modifiers*$_{opt}$ `struct` *identifier* *struct-interfaces*$_{opt}$ *struct-body* `;`$_{opt}$

A *struct-declaration* consists of an optional set of *attributes* (§24), followed by an optional set of *struct-modifiers* (§18.1.1), followed by the keyword `struct` and an *identifier* that names the struct, followed by an optional *struct-interfaces* specification (§18.1.2), followed by a *struct-body* (§18.1.3), optionally followed by a semicolon.

18.1.1 Struct modifiers

A *struct-declaration* may optionally include a sequence of struct modifiers:

> *struct-modifiers:*
>   *struct-modifier*
>   *struct-modifiers* *struct-modifier*
>
> *struct-modifier:*
>   `new`
>   `public`
>   `protected`
>   `internal`
>   `private`

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration (§17.1.1).

18.1.2 Struct interfaces

A struct declaration may include a *struct-interfaces* specification, in which case the struct is said to implement the given interface types.

> *struct-interfaces:*
>   `:` *interface-type-list*

Interface implementations are discussed further in §20.4.

18.1.3 Struct body

The *struct-body* of a struct defines the members of the struct.

struct-body:
  `{` *struct-member-declarations*$_{opt}$ `}`

18.2 Struct members

The members of a struct consist of the members introduced by its *struct-member-declaration*s and the members inherited from the type `System.ValueType`.

struct-member-declarations:
  struct-member-declaration
  struct-member-declarations struct-member-declaration

struct-member-declaration:
  constant-declaration
  field-declaration
  method-declaration
  property-declaration
  event-declaration
  indexer-declaration
  operator-declaration
  constructor-declaration
  static-constructor-declaration
  type-declaration

Except for the differences noted in §18.3, the descriptions of class members provided in §17.2 through §17.11 apply to struct members as well.

18.3 Class and struct differences

18.3.1 Value semantics

Structs are value types (§11.1) and are said to have value semantics. Classes, on the other hand, are reference types (§11.2) 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.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. Furthermore, because structs are not reference types, it is not possible for values of a struct type to be `null`.

[*Example:* 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. *end example*]

18.3.2 Inheritance

All struct types implicitly inherit from `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 the type `System.ValueType`.

18.3.3 Assignment

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

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

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 §14.13.1.

18.3.4 Default values

As described in §12.2, 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`.

[*Example:* Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero. *end example*]

The default value of a struct corresponds to the value returned by the default constructor of the struct (§11.1.1). 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`.

[*Note:* Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;
struct KeyValuePair
{
   string key;
   string value;
```

```
        public KeyValuePair(string key, string value) {
            if (key == null || value == null) throw new ArgumentException();
            this.key = key;
            this.value = value;
        }
    }
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state. *end note*]

18.3.5 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. [*Note:* Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct. *end note*]

For further details on boxing and unboxing, see §11.3.

18.3.6 Meaning of `this`

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class.

Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type, 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.

18.3.7 Field initializers

As described in §18.3.4, 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. [*Example:* As such, the following example results in one or more compile-time errors:

```
    struct Point
    {
        public int x = 1;  // Error, initializer not permitted
        public int y = 1;  // Error, initializer not permitted
    }
```

end example]

This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

18.3.8 Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor, which always returns the value that results from setting all value type fields to their default value and all reference type fields to `null` (§11.1.1). A struct can declare instance constructors having parameters. [*Example:* 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. *end example*]

A struct instance constructor is not permitted to include a constructor initializer of the form `base(…)`.

The `this` variable of a struct instance constructor corresponds to an `out` parameter of the struct type, and similar to an `out` parameter, `this` must be definitely assigned (§12.3) at every location where the constructor returns. [*Example:* Consider the instance constructor implementation below:

```
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.

end example]

18.3.9 Destructors

A struct is not permitted to declare a destructor.

18.4 Struct examples

This whole clause is informative.

18.4.1 Database integer type

The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```
using System;
public struct DBInt
{
   // The Null member represents an unknown DBInt value.
   public static readonly DBInt Null = new DBInt();
```

```
      // When the defined field is true, this DBInt represents a known value
      // 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;
    }
}
```

18.4.2 Database boolean type

The `DBBool` struct below implements a three-valued logical type. The possible values of this type are `DBBool.True`, `DBBool.False`, and `DBBool.Null`, where the `Null` member indicates an unknown value. Such three-valued logical types are commonly used in databases.

```
using System;
public struct DBBool
{
   // The three possible DBBool values.
   public static readonly DBBool Null = new DBBool(0);
   public static readonly DBBool False = new DBBool(-1);
   public static readonly DBBool True = new DBBool(1);
   // Private field that stores –1, 0, 1 for False, Null, True.
   sbyte value;
   // Private instance constructor. The value parameter must be –1, 0, or
1.
   DBBool(int value) {
      this.value = (sbyte)value;
   }
   // Properties to examine the value of a DBBool. Return true if this
   // DBBool has the given value, false otherwise.
   public bool IsNull { get { return value == 0; } }
   public bool IsFalse { get { return value < 0; } }
   public bool IsTrue { get { return value > 0; } }
   // Implicit conversion from bool to DBBool. Maps true to DBBool.True
and
   // false to DBBool.False.
   public static implicit operator DBBool(bool x) {
      return x? True: False;
   }
   // Explicit conversion from DBBool to bool. Throws an exception if the
   // given DBBool is Null, otherwise returns true or false.
   public static explicit operator bool(DBBool x) {
      if (x.value == 0) throw new InvalidOperationException();
      return x.value > 0;
   }
   // Equality operator. Returns Null if either operand is Null,
otherwise
   // returns True or False.
   public static DBBool operator ==(DBBool x, DBBool y) {
      if (x.value == 0 || y.value == 0) return Null;
      return x.value == y.value? True: False;
   }
   // 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;
        }
    }
```

End of informative text.

19. 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.

19.1 Array types

An array type is written as a *non-array-type* followed by one or more *rank-specifier*s:

array-type:
 non-array-type rank-specifiers

non-array-type:
 type

rank-specifiers:
 rank-specifier
 rank-specifiers rank-specifier

rank-specifier:
 `[` *dim-separators*$_{opt}$ `]`

dim-separators:
 `,`
 dim-separators `,`

A *non-array-type* is any *type* that is not itself an *array-type*.

The rank of an array type is given by the leftmost *rank-specifier* in the *array-type*: A *rank-specifier* indicates that the array is an array with a rank of one plus the number of “`,`” tokens in the *rank-specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank-specifier*:

- An array type of the form `T[R]` is an array with rank `R` and a non-array element type `T`.
- An array type of the form `T[R]`[R_1]…[R_N] is an array with rank `R` and an element type `T`[R_1]…[R_N].

In effect, the *rank-specifier*s are read from left to right *before* the final non-array element type. [*Example:* The type `int[][,,][,]` is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of `int`. *end example*]

At run-time, a value of an array type can be `null` or a reference to an instance of that array type.

19.1.1 The `System.Array` type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion (§13.1.4) exists from any array type to `System.Array`, and an explicit reference conversion (§13.2.3) exists from `System.Array` to any array type. Note that `System.Array` is not itself an *array-type*. Rather, it is a *class-type* from which all *array-type*s are derived.

At run-time, a value of type `System.Array` can be `null` or a reference to an instance of any array type.

19.2 Array creation

Array instances are created by *array-creation-expression*s (§14.5.10.2) or by field or local variable declarations that include an *array-initializer* (§19.6).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The `System.Array` type is an abstract type that cannot be instantiated.

Elements of arrays created by *array-creation-expression*s are always initialized to their default value (§12.2).

19.3 Array element access

Array elements are accessed using *element-access* expressions (§14.5.6.1) of the form $A[I_1, I_2, \ldots, I_N]$, where `A` is an expression of an array type and each I_X is an expression of type `int`, `uint`, `long`, `ulong`, or of a type that can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement (§15.8.4).

19.4 Array members

Every array type inherits the members declared by the `System.Array` type.

19.5 Array covariance

For any two *reference-type*s `A` and `B`, if an implicit reference conversion (§13.1.4) or explicit reference conversion (§13.2.3) exists from `A` to `B`, then the same reference conversion also exists from the array type `A[R]` to the array type `B[R]`, where `R` is any given *rank-specifier* (but the same for both array types). This relationship is known as ***array covariance***. Array covariance, in particular, means that a value of an array type `A[R]` may actually be a reference to an instance of an array type `B[R]`, provided an implicit reference conversion exists from `B` to `A`.

Because of array covariance, assignments to elements of reference type arrays include a run-time check which ensures that the value being assigned to the array element is actually of a permitted type (§14.13.1). [*Example:* For example:

```
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 of a type that is compatible with the actual element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array. *end example*]

Array covariance specifically does not extend to arrays of *value-type*s. For example, no conversion exists that permits an `int[]` to be treated as an `object[]`.

19.6 Array initializers

Array initializers may be specified in field declarations (§17.4), local variable declarations (§15.5.1), and array creation expressions (§14.5.10.2):

array-initializer:
  `{` *variable-initializer-list*$_{opt}$ `}`
  `{` *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. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, [*Example:* such as:

```
int[] a = {0, 2, 4, 6, 8};
```

end example] it is simply shorthand for an equivalent array creation expression: [*Example:*

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

end example]

For a single-dimensional array, the array initializer 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. [*Example:* 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;
```

end example]

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. [*Example:* The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

end example]

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. [*Example:* Here are some examples:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};     // OK
int[] y = new int[i] {0, 1, 2};     // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};  // Error, length/initializer mismatch
```

Here, the initializer for `y` results in a compile-time error because the dimension length expression is not a constant, and the initializer for `z` results in a compile-time error because the length and the number of elements in the initializer do not agree. *end example*]

20. 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 interfaces that implement the interface.

20.1 Interface declarations

An *interface-declaration* is a *type-declaration* (§16.5) that declares a new interface type.

interface-declaration:
attributes$_{opt}$ *interface-modifiers*$_{opt}$ `interface` *identifier* *interface-base*$_{opt}$ *interface-body* `;`$_{opt}$

An *interface-declaration* consists of an optional set of *attributes* (§24), followed by an optional set of *interface-modifiers* (§20.1.1), followed by the keyword `interface` and an *identifier* that names the interface, optionally followed by an optional *interface-base* specification (§20.1.2), followed by a *interface-body* (§20.1.3), optionally followed by a semicolon.

20.1.1 Interface modifiers

An *interface-declaration* may optionally include a sequence of interface modifiers:

interface-modifiers:
interface-modifier
interface-modifiers *interface-modifier*

interface-modifier:
`new`
`public`
`protected`
`internal`
`private`

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The `new` modifier is only permitted on nested interfaces. It specifies that the interface hides an inherited member by the same name, as described in §17.2.2.

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 (§10.5.1).

20.1.2 Base interfaces

An interface can inherit from zero or more interfaces, 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 identifiers.

interface-base:
`:` *interface-type-list*

The explicit base interfaces of an interface must be at least as accessible as the interface itself (§10.5.4). [*Note:* For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface-base* of a `public` interface. *end note*]

It is a compile-time error for an interface to directly or indirectly inherit from itself.

The ***base interfaces*** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. [*Example:* In the 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`. *end example*]

A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

20.1.3 Interface body

The *interface-body* of an interface defines the members of the interface.

interface-body:
  `{` *interface-member-declarations*$_{opt}$ `}`

20.2 Interface members

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

interface-member-declarations:
  interface-member-declaration
  interface-member-declarations interface-member-declaration

interface-member-declaration:
  interface-method-declaration
  interface-property-declaration
  interface-event-declaration
  interface-indexer-declaration

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interface members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

[*Example:* 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. *end example*]

An *interface-declaration* creates a new declaration space (§10.3), and the *interface-member-declarations* immediately contained by the *interface-declaration* introduce new members into this declaration space. The following rules apply to *interface-member-declaration*s:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature (§10.6) of a method must differ from the signatures of all other methods declared in the same interface.
- 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 §10.7.1.2.

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.

20.2.1 Interface methods

Interface methods are declared using *interface-method-declaration*s:

interface-method-declaration:
    attributes$_{opt}$ `new`$_{opt}$ *return-type* *identifier* `(` *formal-parameter-list$_{opt}$* `)` `;`

The *attributes*, *return-type*, *identifier*, and *formal-parameter-list* of an interface method declaration have the same meaning as those of a method declaration in a class (§17.5). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

20.2.2 Interface properties

Interface properties are declared using *interface-property-declaration*s:

interface-property-declaration:
    attributes$_{opt}$ `new`$_{opt}$ *type* *identifier* `{` *interface-accessors* `}`

interface-accessors:
    attributes$_{opt}$ `get` `;`
    attributes$_{opt}$ `set` `;`
    attributes$_{opt}$ `get` `;` *attributes$_{opt}$* `set` `;`
    attributes$_{opt}$ `set` `;` *attributes$_{opt}$* `get` `;`

The *attributes, type*, and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class (§17.6).

The accessors of an interface property declaration correspond to the accessors of a class property declaration (§17.6.2), 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.

20.2.3 Interface events

Interface events are declared using *interface-event-declarations*:

> *interface-event-declaration:*
>   *attributes*$_{opt}$ `new`$_{opt}$ `event` *type* *identifier* `;`

The *attributes, type,* and *identifier* of an interface event declaration have the same meaning as those of an event declaration in a class (§17.7).

20.2.4 Interface indexers

Interface indexers are declared using *interface-indexer-declarations*:

> *interface-indexer-declaration:*
>   *attributes*$_{opt}$ `new`$_{opt}$ *type* `this` `[` *formal-parameter-list* `]` `{` *interface-accessors* `}`

The *attributes*, *type*, and *formal-parameter-list* of an interface indexer declaration have the same meaning as those of an indexer declaration in a class (§17.8).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (§17.8), 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.

20.2.5 Interface member access

Interface members are accessed through member access (§14.5.4) and indexer access (§14.5.6.2) expressions of the form `I.M` and `I[A]`, where `I` is an instance of an interface type, `M` is a method, property, or event of 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 (§14.3), method invocation (§14.5.5.1), and indexer access (§14.5.6.2) 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.

[*Example:* 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 (§14.3) of `Count` in `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time. *end example*]

[*Example:* In the 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);                // Error, both Add methods are applicable
      n.Add(1.0);              // Ok, only IDouble.Add is applicable
      ((IInteger)n).Add(1);    // Ok, only IInteger.Add is a candidate
      ((IDouble)n).Add(1);     // Ok, only IDouble.Add is a candidate
   }
}
```

the invocation `n.Add(1)` is ambiguous because a method invocation (§14.5.5.1) requires all overloaded candidate methods to be declared in the same type. However, the invocation `n.Add(1.0)` is permitted because only `IDouble.Add` is applicable. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity. *end example*]

[*Example:* 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`. *end example*]

20.3 Fully qualified interface member names

An interface member is sometimes referred to by its ***fully qualified name***. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. [*Example:* 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`. *end example*]

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. [*Example:* For example

```
namespace System
{
   public interface ICloneable
   {
      object Clone();
   }
}
```

Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`. *end example*]

20.4 Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct implements an interface, the interface identifier is included in the base class list of the class or struct. [*Example:* For example:

```
interface ICloneable
{
   object Clone();
}
interface IComparable
{
   int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
   public object Clone() {…}
   public int CompareTo(object other) {…}
}
```

end example]

A class or struct that implements an interface also implicitly implements all of the interface’s base interfaces. This is true even if the class or struct doesn’t explicitly list all base interfaces in the base class list. [*Example:* 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`. *end example*]

20.4.1 Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare ***explicit interface member implementations***. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. [*Example:* For example

```
interface ICloneable
{
   object Clone();
}
interface IComparable
{
   int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
   object ICloneable.Clone() {…}
   int IComparable.CompareTo(object other) {…}
}
```

Here, `ICloneable.Clone` and `IComparable.CompareTo` are explicit interface member implementations. *end example*]

[*Example:* 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);
   }
}
```

end example]

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. [*Example:* Thus, in the following class

```
class Shape: ICloneable
{
   object ICloneable.Clone() {…}
   int IComparable.CompareTo(object other) {…}  // invalid
}
```

the declaration of `IComparable.CompareTo` results in a compile-time error because `IComparable` is not listed in the base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the declarations

```
class Shape: ICloneable
{
   object ICloneable.Clone() {…}
}
class Ellipse: Shape
{
   object ICloneable.Clone() {…} // invalid
}
```

the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is not explicitly listed in the base class list of `Ellipse`. *end example*]

The fully qualified name of an interface member must reference the interface in which the member was declared. [*Example:* Thus, in the declarations

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
class TextBox: ITextBox
{
   void IControl.Paint() {…}
   void ITextBox.SetText(string text) {…}
}
```

the explicit interface member implementation of `Paint` must be written as `IControl.Paint`. *end example*]

20.4.2 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`.

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.

[*Example:* 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. *end example*]

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. [*Example:* 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. *end example*]

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. [*Example:* 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() {…}
}
```

end example]

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. [*Example:* In the 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`. *end example*]

The members of a base class participate in interface mapping. [*Example:* 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`. *end example*]

20.4.3 Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly ***re-implementing*** an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. [*Example:* 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();
```

end example]

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. [*Example:* 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();
```

end example]

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. [*Example:* 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. *end example*]

20.4.4 Interface re-implementation

A class that inherits an interface implementation is permitted to ***re-implement*** the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. [*Example:* 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`. *end example*]

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. [*Example:* 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`. *end example*]

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. [*Example:* 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`. *end example*]

20.4.5 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. [*Example:* 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`. *end example*]

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. [*Example:* 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`. *end example*]

21. Enums

An ***enum type*** is a distinct type that declares a set of named constants. [*Example:* The example

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

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`. *end example*]

21.1 Enum declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and defines the name, accessibility, underlying type, and members of the enum.

enum-declaration:
    attributes$_{opt}$ *enum-modifiers*$_{opt}$ `enum` *identifier* *enum-base*$_{opt}$ *enum-body* `;`$_{opt}$

enum-base:
    `:` *integral-type*

enum-body:
    `{` *enum-member-declarations*$_{opt}$ `}`
    `{` *enum-member-declarations* `,` `}`

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:* `char` cannot be used as an underlying type. *end note*] An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

[*Example:* The example

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

declares an enum with an underlying type of `long`. *end example*] [*Note:* A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future. *end note*]

21.2 Enum modifiers

An *enum-declaration* may optionally include a sequence of enum modifiers:

enum-modifiers:
    enum-modifier
    enum-modifiers *enum-modifier*

enum-modifier:
`new`
`public`
`protected`
`internal`
`private`

It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

The modifiers of an enum declaration have the same meaning as those of a class declaration (§17.1.1). Note, however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

21.3 Enum members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

enum-member-declarations:
enum-member-declaration
enum-member-declarations `,` *enum-member-declaration*

enum-member-declaration:
attributes$_{opt}$ *identifier*
attributes$_{opt}$ *identifier* `=` *constant-expression*

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. [*Example:* The example

```
enum Color: uint
{
   Red = -1,
   Green = -2,
   Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `–3` are not in the range of the underlying integral type `uint`. *end example*]

Multiple enum members may share the same associated value. [*Example:* The example

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

   Max = Blue
}
```

shows an enum that has two enum members—`Blue` and `Max`—that have the same associated value. *end example*]

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant-expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is zero.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by one. This increased value must be within the range of values that can be represented by the underlying type.

[*Example:* The example

```
using System;

enum Color
{
   Red,
   Green = 10,
   Blue
}

class Test
{
   static void Main() {
      Console.WriteLine(StringFromColor(Color.Red));
      Console.WriteLine(StringFromColor(Color.Green));
      Console.WriteLine(StringFromColor(Color.Blue));
   }

   static string StringFromColor(Color c) {
      switch (c) {
         case Color.Red:
            return String.Format("Red = {0}", (int) c);

         case Color.Green:
            return String.Format("Green = {0}", (int) c);

         case Color.Blue:
            return String.Format("Blue = {0}", (int) c);

         default:
            return "Invalid color";
      }
   }
}
```

prints out the enum member names and their associated values. The output is:

```
Red = 0
Green = 10
Blue = 11
```

for the following reasons:

- the enum member `Red` is automatically assigned the value zero (since it has no initializer and is the first enum member);
- the enum member `Green` is explicitly given the value `10`;
- and the enum member `Blue` is automatically assigned the value one greater than the member that textually precedes it.

end example]

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

[*Example:* The example

```
enum Circular
{
   A = B,
   B
}
```

results in a compile-time error because the declarations of `A` and `B` are circular. `A` depends on `B` explicitly, and `B` depends on `A` implicitly. *end example*]

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member 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.

21.4 Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion (§13.2.2) 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 §21.3). 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: `==`, `!=`, `<`, `>`, `<=`, `>=` (§14.9.5), `+` (§14.7.4), `-` (§14.7.5), `^`, `&`, `|` (§14.10.2), `~` (§14.6.4), `++`, `--` (§14.5.9 and §14.6.5), and `sizeof` (§25.5.4).

Every enum type automatically derives from the class `System.Enum`. Thus, inherited methods and properties of this class can be used on values of an enum type.

22. Delegates

[*Note:* 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. *end note*]

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates 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. Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate instance's methods with that set of arguments.

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 (§22.1) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

22.1 Delegate declarations

A *delegate-declaration* is a *type-declaration* (§16.5) that declares a new delegate type.

delegate-declaration:
    attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type* *identifier*
        `(` *formal-parameter-list*$_{opt}$ `)` `;`

delegate-modifiers:
    delegate-modifier
    delegate-modifiers *delegate-modifier*

delegate-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

The `new` modifier is 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 §17.2.2.

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 (§10.5.1).

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. A method and a delegate type are ***compatible*** if both of the following are true:

- They have the same number or parameters, with the same types, in the same order, with the same parameter modifiers.
- Their *return-type*s are the same.

Delegate types in C# are name equivalent, not structurally equivalent. [*Note*: However, instances of two distinct but structurally equivalent delegate types may compare as equal (§14.9.8). *end note*] Specifically, two different delegate types that have the same parameter lists and return type are considered **different** delegate types. [*Example:* For example:

```
delegate int D1(int i, double d);
class A
{
   public static int M1(int a, double b) {…}
}
class B
{
   delegate int D2(int c, double d);
   public static int M1(int f, double g) {…}
   public static void M2(int k, double l) {…}
   public static int M3(int g) {…}
   public static void M4(int g) {…}
}
```

The delegate types `D1` and `D2` are both compatible with the methods `A.M1` and `B.M1`, since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The delegate types `D1` and `D2` are incompatible with the methods `B.M2`, `B.M3`, and `B.M4`, since they have different return types or parameter lists. *end example*]

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 (§22.2) 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 `+` (§14.7.4) and `+=` operators (§14.13.2). A delegate can be removed from a combination of delegates, using the binary `-` (§14.7.5) and `-=` operators (§14.13.2). Delegates can be compared for equality (§14.9.8).

[*Example:* The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);
class Test
{
   public static void M1(int i) {…}
   public static void M2(int i) {…}
}
class Demo
{
   static void Main() {
      D cd1 = new D(Test.M1); // M1
      D cd2 = new D(Test.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 §22.3. *end example*]

22.2 Delegate instantiation

An instance of a delegate is created by a *delegate-creation-expression* (§14.5.10.3). 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

[*Example:* For example:

```
delegate void D(int x);
class Test
{
   public static void M1(int i)  {…}
   public void M2(int i)      {…}
}
class Demo
{
   static void Main() {
      D cd1 = new D(Test.M1); // static method
      Test t = new Test();
      D cd2 = new D(t.M2);    // instance method
      D cd3 = new D(cd2);     // another delegate
   }
}
```

end example]

Once instantiated, delegate instances always refer to the same target object and method. [*Note:* Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged. *end note*]

22.3 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 §14.5.5.2 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 (§17.5.1.2), 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`.

[*Example:* The following example shows how to instantiate, combine, remove, and invoke delegates:

```
using System;
delegate void D(int x);
class Test
{
   public static void M1(int i) {
      Console.WriteLine("Test.M1: " + i);
   }

   public static void M2(int i) {
      Console.WriteLine("Test.M2: " + i);
   }

   public void M3(int i) {
      Console.WriteLine("Test.M3: " + i);
   }
}

class Demo
{
   static void Main() {
      D cd1 = new D(Test.M1);
      cd1(-1);     // call M1

      D cd2 = new D(Test.M2);
      cd2(-2);     // call M2

      D cd3 = cd1 + cd2;
      cd3(10);     // call M1 then M2


      cd3 += cd1;
      cd3(20);     // call M1, M2, then M1

      Test t = new Test();
      D cd4 = new D(t.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
//    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:

```
Test.M1: -1
Test.M2: -2
```

```
    Test.M1: 10
    Test.M2: 10

    Test.M1: 20
    Test.M2: 20
    Test.M1: 20

    Test.M1: 30
    Test.M2: 30
    Test.M1: 30
    Test.M3: 30

    Test.M1: 40
    Test.M2: 40
    Test.M3: 40

    Test.M1: 50
    Test.M2: 50

    Test.M1: 60
    Test.M1: 60
```

end example]

23. Exceptions

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application-level error conditions. [*Note:* The exception mechanism is 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 (§15.10) 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.

end note]

23.1 Causes of exceptions

Exception can be thrown in two different ways.

- A `throw` statement (§15.9.5) 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 (§14.7.2) throws a `System.DivideByZeroException` if the denominator is zero. See §23.4 for a list of the various exceptions that can occur in this way.

23.2 The `System.Exception` class

The `System.Exception` class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- `Message` is a read-only property of type `string` that contains a human-readable description of the reason for the exception.
- `InnerException` is a read-only property of type `Exception`. If its value is non-`null`, it refers to the exception that caused the current exception. (That is, the current exception was raised in a catch block handling the type `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`.

23.3 How exceptions are handled

Exceptions are handled by a `try` statement (§15.10).

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 (§17.11) 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.

23.4 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.

24. Attributes

[*Note:* 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`. *end note*]

C# enables programmers to invent new kinds of declarative information, called ***attributes.*** Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. [*Note:* For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation. *end note*]

Attributes are defined through the declaration of attribute classes (§24.1), which may have positional and named parameters (§24.1.2). Attributes are attached to entities in a C# program using attribute specifications (§24.2), and can be retrieved at run-time as attribute instances (§24.3).

24.1 Attribute classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an ***attribute class***. The declaration of an attribute class defines a new kind of attribute that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

24.1.1 Attribute usage

The attribute `AttributeUsage` (§24.4.1) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter (§24.1.2) that enables an attribute class to specify the kinds of declarations on which it can be used. [*Example:* The example

```
using System;
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class-declaration*s and *interface-declaration*s only. The example

```
[Simple] class Class1 {…}
[Simple] interface Interface1 {…}
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the short name `Simple`. Thus, the example above is semantically equivalent to the following

```
[SimpleAttribute] class Class1 {…}
[SimpleAttribute] interface Interface1 {…}
```

end example]

`AttributeUsage` has a named parameter (§24.1.2), 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 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 class is a ***single-use attribute class***, and can be specified at most once on an entity.

[*Example:* The example

```
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute {
   public AuthorAttribute(string name) {
      this.name = name;
   }
   public string Name { get { return name;} }
   private string name;
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1 {…}
```

shows a class declaration with two uses of the `Author` attribute. *end example*]

`AttributeUsage` has another named parameter (§24.1.2), 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 or it is unspecified, then that attribute is not inherited.

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 { … }
```

24.1.2 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.

[*Example:* The example

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
   public HelpAttribute(string url) {  // url is a positional parameter
      …
   }
   public string Topic {   // Topic is a named parameter
      get {…}
      set {…}
   }
   public string Url { get {…} }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter (`string url`) and one named parameter (`string Topic`). Although it is non-static and public, the property `Url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/…/Class1.htm")]
class Class1 {
}
```

```
    [Help("http://www.mycompany.com/…/Misc.htm", Topic ="Class2")]
    class Class2 {
    }
```

end example]

24.1.3 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`, `short`, `string`.
- 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.
- Single-dimensional arrays of the above types.

24.2 Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly) and for *type-declaration*s (§16.5), *class-member-declaration*s (§17.2), *interface-member-declaration*s (§20.2), *enum-member-declaration*s (§21.1), *accessor-declaration*s (§17.6.2), *event-accessor-declaration*s (§17.7), and *formal-parameter-list*s (§17.5.1).

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`

attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections attribute-section

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `,]`

attribute-target-specifier:
    attribute-target `:`

attribute-target:
`field`
`event`
`method`
`param`
`property`
`return`
`type`

attribute-list:
attribute
attribute-list `,` *attribute*

attribute:
attribute-name *attribute-arguments*$_{opt}$

attribute-name:
type-name

attribute-arguments:
`(` *positional-argument-list*$_{opt}$ `)`
`(` *positional-argument-list* `,` *named-argument-list* `)`
`(` *named-argument-list* `)`

positional-argument-list:
positional-argument
positional-argument-list `,` *positional-argument*

positional-argument:
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. [*Example:* 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. *end example*]

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute-target-specifier*. When an attribute is placed at the global level, a *global-attribute-target-specifier* is required. In all other locations, a reasonable default is applied, but an *attribute-target-specifier* can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, *attribute-target-specifier*s can be omitted except at the global level. The potentially ambiguous contexts are resolved as follows:

- An attribute specified 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 of this declaration. In the absence of an *attribute-target-specifier*, the attribute applies to the operator. The presence of the `type` *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 declaration. 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.
- 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.

An implementation may accept other attribute target specifiers, the purpose of which is implementation-defined. However, an implementation that does not recognize such a target, shall issue a warning.

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 shall be issued. If the *attribute-name* is spelled using a verbatim identifier (§9.4.2), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. [*Example:* 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 {}
```

end example]

It is a compile-time error to use a single-use attribute class more than once on the same entity. [*Example:* The example

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
   string value;

   public HelpStringAttribute(string value) {
      this.value = value;
   }

   public string Value { get {…} }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`. *end example*]

An expression `E` is an *attribute-argument-expression* if all of the following statements are true:

- The type of `E` is an attribute parameter type (§24.1.3).
- 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*.

[*Example:* For example:

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class MyAttribute: Attribute
{
   public int P1 {
      get {…}
      set {…}
   }
   public Type P2 {
      get {…}
      set {…}
   }
   public object P3 {
      get {…}
      set {…}
   }
}
[My(P1 = 1234, P3 = new int[]{1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

end example]

24.3 Attribute instances

An ***attribute instance*** is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

24.3.1 Compilation of an attribute

The compilation of an *attribute* with attribute class `T`, *positional-argument-list* `P` and *named-argument-list* `N`, consists of the following steps:

- 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 on `T` that can be invoked at run-time. Call this instance constructor `C`.
- 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`.

24.3.2 Run-time retrieval of an attribute instance

Compilation of an *attribute* yields an attribute class `T`, an instance constructor `C` on `T`, a *positional-argument-list* `P`, and a *named-argument-list* `N`. Given this information, an attribute instance can be retrieved at run-time using the following steps:

- 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 of `T`. Call this instance `O`.
- 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 the value `Value`.
- Otherwise, `Name` identifies a property on `O`. Set this property to the value `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`.

24.4 Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

- `System.AttributeUsageAttribute` (§24.4.1), which is used to describe the ways in which an attribute class can be used.
- `System.ConditionalAttribute` (§24.4.2), which is used to define conditional methods.
- `System.ObsoleteAttribute` (§24.4.3), which is used to mark a member as obsolete.

24.4.1 The `AttributeUsage` attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either directly or indirectly. Otherwise, a compile-time error occurs.

[*Note:* For an example of using this attribute, see §24.1.1. *end note*]

24.4.2 The `Conditional` attribute

The attribute `Conditional` enables the definition of ***conditional methods***. 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 is omitted.

A conditional method is subject to the following restrictions:

- The conditional method must be a method in a *class-declaration*. A compile-time error occurs if the `Conditional` attribute is specified on an interface method.
- 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*.

[*Example:* The example

```
#define DEBUG
```

```
using System;
using System.Diagnostics;
class Class1
{
   [Conditional("DEBUG")]
   public static void M() {
      Console.WriteLine("Executed Class1.M");
   }
}
class Class2
{
   public static void Test() {
      Class1.M();
   }
}
```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`. *end example*]

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. [*Example:* In the example

```
// Begin class1.cs
   using System;
   using System.Diagnostics;
   class Class1
   {
      [Conditional("DEBUG")]
      public static void F() {
         Console.WriteLine("Executed Class1.F");
      }
   }
// End class1.cs

// Begin class2.cs
   #define DEBUG
   class Class2
   {
      public static void G() {
         Class1.F();          // F is called
      }
   }
// End class2.cs

// Begin class3.cs
   #undef DEBUG
   class Class3
   {
      public static void H() {
         Class1.F();          // F is not called
      }
   }
// End class3.cs
```

the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to `F` in `Class3` is omitted. *end example*]

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through `base`, of the form `base.M`, are subject to the normal conditional method call rules. [*Example:* In the example

```
// Begin class1.cs
   using System;
   using System.Diagnostics;
   class Class1
   {
      [Conditional("DEBUG")]
      public virtual void M() {
         Console.WriteLine("Class1.M executed");
      }
   }
// End class1.cs

// Begin class2.cs
   using System;
   class Class2: Class1
   {
      public override void M() {
         Console.WriteLine("Class2.M executed");
         base.M();                  // base.M is not called!
      }
   }
// End class2.cs

// Begin class3.cs
   #define DEBUG
   using System;
   class Class3
   {
      public static void Test() {
         Class2 c = new Class2();
         c.M();                     // M is called
      }
   }
// End class3.cs
```

`Class2` includes a call to the `M` defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol `DEBUG`, which is undefined. Thus, the method writes to the console “`Class2.M executed`” only. Judicious use of *pp-declaration*s can eliminate such problems. *end example*]

24.4.3 The `Obsolete` attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
using System;
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct |
   AttributeTargets.Enum | AttributeTargets.Interface |
   AttributeTargets.Delegate | AttributeTargets.Method |
   AttributeTargets.Constructor | AttributeTargets.Property |
   AttributeTargets.Field | AttributeTargets.Event)]
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, then the compiler shall issue a warning or error in order to alert the developer, so the offending code can be fixed. Specifically, the compiler shall issue a warning if no error parameter is provided, or if the error parameter is provided and has the value false. The compiler shall issue a compile-time error if the error parameter is specified and has the value true.

[*Example*: 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.”

end example]

25. Unsafe code

An implementation that does not support unsafe code is required to diagnose any usage of the keyword `unsafe`.

This remainder of this clause is conditionally normative.

[*Note:* The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language it is simply not possible to have an uninitialized variable, a “dangling” pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm may not be possible or practical without access to pointers. To address this need, C# provides the ability to write ***unsafe code***.

In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is in fact a “safe” feature from the perspective of both developers and users. Unsafe code must be clearly marked with the modifier `unsafe`, so developers can’t possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment. *end note*]

25.1 Unsafe contexts

The unsafe features of C# are available only in *unsafe contexts*. An unsafe context is introduced by including an `unsafe` modifier in the declaration of a type or member, or by employing an *unsafe-statement*:

- A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context.
- A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or static constructor may include an `unsafe` modifier, in which case, the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe-statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar extensions are shown below. For brevity, ellipses (...) are used to represent productions that appear in preceding chapters.

class-modifier:
  ...
  `unsafe`

struct-modifier:
  ...
  `unsafe`

interface-modifier:
...
`unsafe`

delegate-modifier:
...
`unsafe`

field-modifier:
...
`unsafe`

method-modifier:
...
`unsafe`

property-modifier:
...
`unsafe`

event-modifier:
...
`unsafe`

indexer-modifier:
...
`unsafe`

operator-modifier:
...
`unsafe`

constructor-modifier:
...
`unsafe`

destructor-declaration:
attributes$_{opt}$ `extern`$_{opt}$ `unsafe`$_{opt}$ `~` *identifier* `( )` *destructor-body*
attributes$_{opt}$ `unsafe`$_{opt}$ `extern`$_{opt}$ `~` *identifier* `( )` *destructor-body*

static-constructor-declaration:
attributes$_{opt}$ `extern`$_{opt}$ `unsafe`$_{opt}$ `static` *identifier* `( )` *static-constructor-body*
attributes$_{opt}$ `unsafe`$_{opt}$ `extern`$_{opt}$ `static` *identifier* `( )` *static-constructor-body*

embedded-statement:
...
unsafe-statement

unsafe-statement:
`unsafe` *block*

[*Example:* In the example

```
public unsafe struct Node
{
   public int Value;
   public Node* Left;
   public Node* Right;
}
```

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written

```
public struct Node
{
   public int Value;
   public unsafe Node* Left;
   public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts. *end example*]

Other than establishing an unsafe context, thus permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. [*Example:* In the example

```
public class A
{
   public unsafe virtual void F() {
      char* p;
      …
   }
}
public class B: A
{
   public override void F() {
      base.F();
      …
   }
}
```

the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier—unless, of course, the `F` method in `B` itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature

```
public unsafe class A
{
   public virtual void F(char* p) {…}
}
public class B: A
{
   public unsafe override void F(char* p) {…}
}
```

Here, because `F`'s signature includes a pointer type, it can only be written in an unsafe context. However, the unsafe context can be introduced by either making the entire class unsafe, as is the case in A, or by including an `unsafe` modifier in the method declaration, as is the case in B. *end example*]

25.2 Pointer types

In an unsafe context, a *type* (§11) may be a *pointer-type* as well as a *value-type* or a *reference-type*.

type:
    value-type
    reference-type
    pointer-type

A *pointer-type* is written as an *unmanaged-type* or the keyword `void`, followed by a `*` token:

pointer-type:
    unmanaged-type `*`
    `void *`

unmanaged-type:
    type

The type specified before the `*` in a pointer type is called the ***referent type*** of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector—the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must be an *unmanaged-type*.

An *unmanaged-type* is any type that isn't a *reference-type* and doesn't contain *reference-type* fields at any level of nesting. In other words, an *unmanaged-type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- Any *enum-type*.
- Any *pointer-type*.
- Any user-defined *struct-type* that contains fields of *unmanaged-type*s only.

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

[*Example:* Some examples of pointer types are given in the table below:

Example	**Description**
`byte*`	Pointer to `byte`
`char*`	Pointer to `char`
`int**`	Pointer to pointer to `int`
`int*[]`	Single-dimensional array of pointers to `int`
`void*`	Pointer to unknown type

end example]

For a given implementation, all pointer types must have the same size and representation.

[*Note:* Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example:

```
int* pi, pj;   // NOT as int *pi, *pj;
```

end note]

The value of a pointer having type `T*` represents the *address* of a variable of type `T`. The pointer indirection operator `*` (§25.5.1) may be used to access this variable. For example, given a variable `P` of type `int*`, the expression `*P` denotes the `int` variable found at the address contained in P.

Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and unboxing (§11.3) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in §25.4.

A *pointer-type* may be used as the type of a volatile field (§17.4.3).

[*Note:* Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior, since the pointer may well be set to point to a local variable which no longer exists when the called method returns, or the fixed object to which it used to point, is no longer fixed. For example:

```
using System;
class Test
{
   static int value = 20;

   unsafe static void F(out int* pi1, ref int* pi2) {
      int i = 10;
      pi1 = &i;

      fixed (int* pj = &value) {
         // ...
         pi2 = pj;
      }
   }

   static void Main() {
      int i = 10;
      unsafe {
         int* px1;
         int* px2 = &i;

         F(out px1, ref px2);
         Console.WriteLine("*px1 = {0}, *px2 = {1}",
            *px1, *px2);   // undefined behavior
      }
   }
}
```

end note]

A method can return a value of some type, and that type can be a pointer. [*Example:* For example, when given a pointer to a contiguous sequence of `int`s, that sequence's element count, and some other `int` value, the following method returns the address of that value in that sequence, if a match occurs; otherwise it returns `null`:

```
unsafe static int* Find(int* pi, int size, int value) {
   for (int i = 0; i < size; ++i) {
      if (*pi == value) {
         return pi;
      }
      ++pi;
   }
   return null;
}
```

end example]

In an unsafe context, several constructs are available for operating on pointers:

- The * operator may be used to perform pointer indirection (§25.5.1).
- The -> operator may be used to access a member of a struct through a pointer (§25.5.2).
- The [] operator may be used to index a pointer (§25.5.3).
- The & operator may be used to obtain the address of a variable (§25.5.4).
- The ++ and -- operators may be used to increment and decrement pointers (§25.5.5).
- The + and - operators may be used to perform pointer arithmetic (§25.5.6).
- The ==, !=, <, >, <=, and => operators may be used to compare pointers (§25.5.7).
- The `stackalloc` operator may be used to allocate memory from the call stack (§25.7).

- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (§25.6).

25.3 Fixed and moveable variables

The address-of operator (§25.5.4) and the `fixed` statement (§25.6) divide variables into two categories: ***Fixed variables*** and ***moveable variables***.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

The `&` operator (§25.5.4) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can only be obtained using a `fixed` statement (§25.6), and that address remains valid only for the duration of that `fixed` statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple-name* (§14.5.2) that refers to a local variable or a value parameter.
- A variable resulting from a *member-access* (§14.5.4) of the form `V.I`, where `V` is a fixed variable of a *struct-type*.
- A variable resulting from a *pointer-indirection-expression* (§25.5.1) of the form `*P`, a *pointer-member-access* (§25.5.2) of the form `P->I`, or a *pointer-element-access* (§25.5.3) of the form `P[E]`.

All other variables are classified as moveable variables.

Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

25.4 Pointer conversions

In an unsafe context, the set of available implicit conversions (§13.1) is extended to include the following implicit pointer conversions:

- From any *pointer-type* to the type `void*`.
- From the null type to any *pointer-type*.

Additionally, in an unsafe context, the set of available explicit conversions (§13.2) is extended to include the following explicit pointer conversions:

- From any *pointer-type* to any other *pointer-type*.
- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer-type*.
- From any *pointer-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Finally, in an unsafe context, the set of standard implicit conversions (§13.3.1) includes the following pointer conversion:

- From any *pointer-type* to the type `void*`.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept “correctly aligned” is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which, in turn, is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

[*Example:* Consider the following case in which a variable having one type is accessed via a pointer to a different type:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;      // undefined
*pi = 123456;     // undefined
```

end example]

When a pointer type is converted to a pointer to `byte`, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. [*Example:* For example, the following method displays each of the eight bytes in a `double` as a hexadecimal value:

```
using System;
class Test
{
   static void Main() {
      double d = 123.456e23;
      unsafe {
         byte* pb = (byte*)&d;

         for (int i = 0; i < sizeof(double); ++i)
            Console.Write(" {0,2:X}", (uint)(*pb++));
         Console.WriteLine();
      }
   }
}
```

Of course, the output produced depends on endianness. *end example*]

Mappings between pointers and integers are implementation-defined. [*Note:* However, on 32- and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types. *end note*]

25.5 Pointers in expressions

In an unsafe context, an expression may yield a result of a pointer type, but outside an unsafe context it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a compile-time error occurs if any *simple-name* (§14.5.2), *member-access* (§14.5.4), *invocation-expression* (§14.5.5), or *element-access* (§14.5.6) is of a pointer type.

In an unsafe context, the *primary-no-array-creation-expression* (§14.5) and *unary-expression* (§14.6) productions permit the following additional constructs:

primary-no-array-creation-expression:
   …
   pointer-member-access
   pointer-element-access
   sizeof-expression

unary-expression:
   …
   pointer-indirection-expression
   addressof-expression

These constructs are described in the following sections.

[*Note:* The precedence and associativity of the unsafe operators is implied by the grammar. *end note*]

25.5.1 Pointer indirection

A *pointer-indirection-expression* consists of an asterisk (`*`) followed by a *unary-expression*.

pointer-indirection-expression:
    `*` *unary-expression*

The unary `*` operator denotes *pointer indirection* and is used to obtain the variable to which a pointer points. The result of evaluating `*P`, where `P` is an expression of a pointer type `T*`, is a variable of type `T`. It is a compile-time error to apply the unary `*` operator to an expression of type `void*` or to an expression that isn't of a pointer type.

The effect of applying the unary `*` operator to a `null` pointer is implementation-defined. In particular, there is no guarantee that this operation throws a `System.NullReferenceException`.

If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined. [*Note:* Among the invalid values for dereferencing a pointer by the unary `*` operator are an address inappropriately aligned for the type pointed to (see example in §25.4), and the address of a variable after the end of its lifetime. *end note*]

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned (§12.3.1).

25.5.2 Pointer member access

A *pointer-member-access* consists of a *primary-expression*, followed by a "`->`" token, followed by an *identifier*.

pointer-member-access:
    primary-expression `->` *identifier*

In a pointer member access of the form `P->I`, `P` must be an expression of a pointer type other than `void*`, and `I` must denote an accessible member of the type to which `P` points.

A pointer member access of the form `P->I` is evaluated exactly as `(*P).I`. For a description of the pointer indirection operator (`*`), see §25.5.1. For a description of the member access operator (`.`), see §14.5.4.

[*Example:* In the example

```
struct Point
{
   public int x;
   public int y;

   public override string ToString() {
      return "(" + x + "," + y + ")";
   }
}

using System;
class Test
{
   static void Main() {
      Point point;
      unsafe {
         Point* p = &point;
         p->x = 10;
         p->y = 20;
         Console.WriteLine(p->ToString());
      }
   }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:

```
using System;
class Test
{
   static void Main() {
      Point point;
      unsafe {
         Point* p = &point;
         (*p).x = 10;
         (*p).y = 20;
         Console.WriteLine((*p).ToString());
      }
   }
}
```

end example]

25.5.3 Pointer element access

A *pointer-element-access* consists of a *primary-no-array-creation-expression* followed by an expression enclosed in "`[`" and "`]`".

> *pointer-element-access:*
>   *primary-no-array-creation-expression* `[` *expression* `]`

In a pointer element access of the form `P[E]`, `P` must be an expression of a pointer type other than `void*`, and `E` must be an expression of a type that can be implicitly converted to `int`, `uint`, `long`, or `ulong`.

A pointer element access of the form `P[E]` is evaluated exactly as `*(P + E)`. For a description of the pointer indirection operator (`*`), see §25.5.1. For a description of the pointer addition operator (`+`), see §25.5.6.

[*Example:* In the example

```
class Test
{
   static void Main() {
      unsafe {
         char* p = stackalloc char[256];
         for (int i = 0; i < 256; i++) p[i] = (char)i;
      }
   }
}
```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written:

```
class Test
{
   static void Main() {
      unsafe {
         char* p = stackalloc char[256];
         for (int i = 0; i < 256; i++) *(p + i) = (char)i;
      }
   }
}
```

end example]

The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. [*Note:* This is the same as C and C++. *end note*]

25.5.4 The address-of operator

An *addressof-expression* consists of an ampersand (`&`) followed by a *unary-expression*.

> *addressof-expression:*
>   `&` *unary-expression*

Given an expression `E` which is of a type `T` and is classified as a fixed variable (§25.3), the construct `&E` computes the address of the variable given by E. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a volatile field (§17.4.3), or if `E` denotes a moveable variable. In the last case, a fixed statement (§25.6) can be used to temporarily "fix" the variable before obtaining its address.

The `&` operator does not require its argument to be definitely assigned, but following an `&` operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

[*Example:* In the example

```
using System;
class Test
{
   static void Main() {
      int i;
      unsafe {
         int* p = &i;
         *p = 123;
      }
      Console.WriteLine(i);
   }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`. The assignment to `*p` in effect initializes `i`, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed. *end example*]

[*Note:* The rules of definite assignment for the `&` operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure which is filled in by the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule, redundant initialization of the struct variable would be required. *end note*]

[*Note:* As stated in §14.5.4, outside an instance constructor or static constructor for a struct or class that defines a readonly field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken. *end note*]

25.5.5 Pointer increment and decrement

In an unsafe context, the `++` and `--` operators (§14.5.9 and §14.6.5) can be applied to pointer variables of all types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator ++(T* x);

T* operator --(T* x);
```

The operators produce the same results as `x+1` and `x-1`, respectively (§25.5.6). In other words, for a pointer variable of type `T*`, the `++` operator adds `sizeof(T)` to the address contained in the variable, and the `--` operator subtracts `sizeof(T)` from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

25.5.6 Pointer arithmetic

In an unsafe context, the `+` operator (§14.7.4) and `-` operator (§14.7.5) can be applied to values of all pointer types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);
```

```
T* operator –(T* x, int y);
T* operator –(T* x, uint y);
T* operator –(T* x, long y);
T* operator –(T* x, ulong y);
long operator –(T* x, T* y);
```

Given an expression `P` of a pointer type `T*` and an expression `N` of type `int`, `uint`, `long`, or `ulong`, the expressions `P + N` and `N + P` compute the pointer value of type `T*` that results from adding `N * sizeof(T)` to the address given by P. Likewise, the expression `P – N` computes the pointer value of type `T*` that results from subtracting `N * sizeof(T)` from the address given by P.

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P – Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`. [*Example:* For example:

```
using System;
class Test
{
   static void Main() {
      unsafe {
         int* values = stackalloc int[20];

         int* p = &values[1];
         int* q = &values[15];

         Console.WriteLine("p - q = {0}", p - q);
         Console.WriteLine("q - p = {0}", q - p);
      }
   }
}
```

which produces the output:

```
p - q = -14
q - p = 14
```

end example]

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an implementation-defined fashion, but no exceptions are produced.

25.5.7 Pointer comparison

In an unsafe context, the ==, !=, <, >, <=, and => operators (§14.9) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

25.5.8 The `sizeof` operator

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified as an operand to `sizeof` must be an *unmanaged-type* (§25.2).

sizeof-expression:
   `sizeof ( ` *unmanaged-type* ` )`

The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator yields a constant value as shown in the table below.

Expression	Result
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(bool)`	1

For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

25.6 The `fixed` statement

In an unsafe context, the *embedded-statement* (§15) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

embedded-statement:
 ...
 fixed-statement

fixed-statement:
 `fixed` `(` *pointer-type* *fixed-pointer-declarators* `)` *embedded-statement*

fixed-pointer-declarators:
 fixed-pointer-declarator
 fixed-pointer-declarators `,` *fixed-pointer-declarator*

fixed-pointer-declarator:
 identifier = *fixed-pointer-initializer*

fixed-pointer-initializer:
 `&` *variable-reference*
 expression

Each *fixed-pointer-declarator* declares a local variable of the given *pointer-type* and initializes that local variable with the address computed by the corresponding *fixed-pointer-initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed-pointer-initializer*s occurring to the right of that variable's declaration, and in the *embedded-statement* of the `fixed` statement. A local variable declared by a `fixed` statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the `++` and `--` operators) or pass it as a `ref` or `out` parameter.

A *fixed-pointer-initializer* can be one of the following:

- The token “&” followed by a *variable-reference* (§12.3.3) to a moveable variable (§25.3) of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the `fixed` statement.
- An expression of an *array-type* with elements of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the array expression is null or if the array has zero elements.
- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.

For each address computed by a *fixed-pointer-initializer* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. For example, if the address computed by a *fixed-pointer-initializer* references a field of an object or an element of an array instance, the `fixed` statement guarantees that the containing object instance is not relocated or disposed of during the lifetime of the statement.

It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive beyond execution of those statements. For example, when pointers created by `fixed` statements are passed to external APIs, it is the programmer’s responsibility to ensure that the APIs retain no memory of these pointers.

Fixed objects may cause fragmentation of the heap (because they can’t be moved). For that reason, objects should be fixed only when absolutely necessary and then only for the shortest amount of time possible.[*Example:* The example

```
class Test
{
   static int x;
   int y;

   unsafe static void F(int* p) {
      *p = 1;
   }

   static void Main() {
      Test t = new Test();
      int[] a = new int[10];
      unsafe {
         fixed (int* p = &x) F(p);
         fixed (int* p = &t.y) F(p);
         fixed (int* p = &a[0]) F(p);
         fixed (int* p = a) F(p);
      }
   }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The third and fourth `fixed` statements in the example above produce identical results. In general, for an array instance `a`, specifying `&a[0]` in a `fixed` statement is the same as simply specifying `a`.

Here's another example of the `fixed` statement, this time using `string`:

```
class Test
{
   static string name = "xx";

   unsafe static void F(char* p) {
      for (int i = 0; p[i] != '\0'; ++i)
         Console.WriteLine(p[i]);
   }

   static void Main() {
      unsafe {
         fixed (char* p = name) F(p);
         fixed (char* p = "xx") F(p);
      }
   }
}
```

end example]

In an unsafe context array elements of single-dimensional arrays are stored in increasing index order, starting with index `0` and ending with index `Length – 1`. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

Within a `fixed` statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p` to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored , we can treat an array of any dimension as though it were linear. [*Example:* For example.

```
using System;
class Test
{
   static void Main() {
      int[,,] a = new int[2,3,4];
      unsafe {
         fixed (int* p = a) {
            for (int i = 0; i < a.Length; ++i)  // treat as linear
               p[i] = i;
         }
      }

      for (int i = 0; i < 2; ++i)
         for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 4; ++k)
               Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k,
a[i,j,k]);
            Console.WriteLine();
         }
   }
}
```

which produces the output:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

end example]

[*Example:* In the example

```
class Test
{
   unsafe static void Fill(int* p, int count, int value) {
      for (; count != 0; count--) *p++ = value;
   }

   static void Main() {
      int[] a = new int[100];
      unsafe {
         fixed (int* p = a) Fill(p, 100, -1);
      }
   }
}
```

a `fixed` statement is used to fix an array so its address can be passed to a method that takes a pointer. *end example*]

A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a fixed statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

Modifying objects of managed type through fixed pointers can result in undefined behavior. [*Note:* For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified. *end note*]

[*Note:* The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`. *end note*]

25.7 Stack allocation

In an unsafe context, a local variable declaration (§15.5.1) may include a stack allocation initializer, which allocates memory from the call stack.

local-variable-initializer:
  expression
  array-initializer
  stackalloc-initializer

stackalloc-initializer:
  `stackalloc` *unmanaged-type* `[` *expression* `]`

The *unmanaged-type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a constant-expression that evaluates to a negative value.

A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type (§25.2) and `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is undefined. If E is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a `System.StackOverflowException` is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in catch or finally blocks (§15.10).

[*Note:* There is no way to explicitly free memory allocated using `stackalloc`. *end note*] All stack-allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. [*Note:* This corresponds to the `alloca` function, an extension commonly found C and C++ implementations. *end note*]

[*Example:* In the example

```
using System;
class Test
{
   static string IntToString(int value) {
      int n = value >= 0 ? value : -value;
      unsafe {
         char* buffer = stackalloc char[16];
         char* p = buffer + 16;
         do {
            *--p = (char)(n % 10 + '0');
            n /= 10;
         } while (n != 0);
         if (value < 0) *--p = '-';
         return new string(p, 0, (int)(buffer + 16 - p));
      }
   }

   static void Main() {
      Console.WriteLine(IntToString(12345));
      Console.WriteLine(IntToString(-999));
   }
}
```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns. *end example*]

25.8 Dynamic memory allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. [*Example:* For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

```
using System;
using System.Runtime.InteropServices;

public unsafe class Memory
{
   // Handle for the process heap. This handle is used in all calls to
the
   // HeapXXX APIs in the methods below.

   static int ph = GetProcessHeap();

   // Private instance constructor to prevent instantiation.

   private Memory() {}

   // Allocates a memory block of the given size. The allocated memory is
   // automatically initialized to zero.

   public static void* Alloc(int size) {
      void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
      if (result == null) throw new OutOfMemoryException();
      return result;
   }

   // Copies count bytes from src to dst. The source and destination
   // blocks are permitted to overlap.
```

```
   public static void Copy(void* src, void* dst, int count) {
      byte* ps = (byte*)src;
      byte* pd = (byte*)dst;
      if (ps > pd) {
         for (; count != 0; count--) *pd++ = *ps++;
      }
      else if (ps < pd) {
         for (ps += count, pd += count; count != 0; count--) *--pd = *--
ps;
      }
   }

   // Frees a memory block.

   public static void Free(void* block) {
      if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
   }

   // Re-allocates a memory block. If the reallocation request is for a
   // larger size, the additional region of memory is automatically
   // initialized to zero.

   public static void* ReAlloc(void* block, int size) {
      void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
      if (result == null) throw new OutOfMemoryException();
      return result;
   }

   // Returns the size of a memory block.

   public static int SizeOf(void* block) {
      int result = HeapSize(ph, 0, block);
      if (result == -1) throw new InvalidOperationException();
      return result;
   }

   // Heap API flags

   const int HEAP_ZERO_MEMORY = 0x00000008;

   // Heap API functions

   [DllImport("kernel32")]
   static extern int GetProcessHeap();

   [DllImport("kernel32")]
   static extern void* HeapAlloc(int hHeap, int flags, int size);

   [DllImport("kernel32")]
   static extern bool HeapFree(int hHeap, int flags, void* block);

   [DllImport("kernel32")]
   static extern void* HeapReAlloc(int hHeap, int flags,
      void* block, int size);

   [DllImport("kernel32")]
   static extern int HeapSize(int hHeap, int flags, void* block);
}
```

An example that uses the `Memory` class is given below:

```
class Test
{
   static void Main() {
      unsafe {
         byte* buffer = (byte*)Memory.Alloc(256);
         for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
         byte[] array = new byte[256];
         fixed (byte* p = array) Memory.Copy(buffer, p, 256);
         Memory.Free(buffer);
         for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
      }
   }
}
```

The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256-element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console. *end example*]

End of conditionally normative text.

A. Grammar

This clause is informative.

This appendix contains summaries of the lexical and syntactic grammars found in the main document, and of the grammar extensions for unsafe code. Grammar productions appear here in the same order that they appear in the main document.

A.1 Lexical grammar

input::
  input-section$_{opt}$

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

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

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

input-element::
  whitespace
  comment
  token

A.1.1 Line terminators

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

A.1.2 White space

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

A.1.3 Comments

comment::
  single-line-comment
  delimited-comment

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

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

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

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

delimited-comment::
`/*` *delimited-comment-characters*$_{opt}$ `*/`

delimited-comment-characters::
delimited-comment-character
delimited-comment-characters delimited-comment-character

delimited-comment-character::
not-asterisk
`*` *not-slash*

not-asterisk::
Any Unicode character except `*`

not-slash::
Any Unicode character except `/`

A.1.4 Tokens

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

A.1.5 Unicode character escape sequences

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

A.1.6 Identifiers

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

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

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

identifier-start-character::
letter-character
`_` (the underscore character)

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

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

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

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

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

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

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

A.1.7 Keywords

keyword:: one of

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

A.1.8 Literals

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

boolean-literal::
`true`
`false`

integer-literal::
decimal-integer-literal
hexadecimal-integer-literal

decimal-integer-literal::
decimal-digits *integer-type-suffix*$_{opt}$

decimal-digits::
decimal-digit
decimal-digits *decimal-digit*

decimal-digit:: one of
`0 1 2 3 4 5 6 7 8 9`

integer-type-suffix:: one of
`U u L l UL Ul uL ul LU Lu lU lu`

hexadecimal-integer-literal::
`0x` *hex-digits* *integer-type-suffix*$_{opt}$
`0X` *hex-digits* *integer-type-suffix*$_{opt}$

hex-digits::
hex-digit
hex-digits *hex-digit*

hex-digit:: one of
`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

real-literal::
decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
`.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
decimal-digits *exponent-part* *real-type-suffix*$_{opt}$
decimal-digits *real-type-suffix*

exponent-part::
`e` *sign*$_{opt}$ *decimal-digits*
`E` *sign*$_{opt}$ *decimal-digits*

sign:: one of
`+ -`

real-type-suffix:: one of
`F f D d M m`

character-literal::
`'` *character* `'`

character::
  single-character
  simple-escape-sequence
  hexadecimal-escape-sequence
  unicode-character-escape-sequence

single-character::
  Any character except `'` (`U+0027`), \ (`U+005C`), and *new-line-character*

simple-escape-sequence:: one of
  `\'` `\"` `\\` `\0` `\a` `\b` `\f` `\n` `\r` `\t` `\v`

hexadecimal-escape-sequence::
  `\x` *hex-digit* $\textit{hex-digit}_{opt}$ $\textit{hex-digit}_{opt}$ $\textit{hex-digit}_{opt}$

string-literal::
  regular-string-literal
  verbatim-string-literal

regular-string-literal::
  `"` $\textit{regular-string-literal-characters}_{opt}$ `"`

regular-string-literal-characters::
  regular-string-literal-character
  regular-string-literal-characters *regular-string-literal-character*

regular-string-literal-character::
  single-regular-string-literal-character
  simple-escape-sequence
  hexadecimal-escape-sequence
  unicode-character-escape-sequence

single-regular-string-literal-character::
  Any character except `"` (`U+0022`), \ (`U+005C`), and *new-line-character*

verbatim-string-literal::
  `@"` $\textit{verbatim -string-literal-characters}_{opt}$ `"`

verbatim-string-literal-characters::
  verbatim-string-literal-character
  verbatim-string-literal-characters *verbatim-string-literal-character*

verbatim-string-literal-character::
  single-verbatim-string-literal-character
  quote-escape-sequence

single-verbatim-string-literal-character::
  any character except `"`

quote-escape-sequence::
  `""`

null-literal::
  `null`

A.1.9 Operators and punctuators

operator-or-punctuator:: one of

```
{    }    [    ]    (    )    .    ,    :    ;
+    -    *    /    %    &    |    ^    !    ~
=    <    >    ?    ++   --   &&   ||   <<   >>
==   !=   <=   >=   +=   -=   *=   /=   %=   &=
|=   ^=   <<=  >>=  ->
```

A.1.10 Pre-processing directives

pp-directive::
    pp-declaration
    pp-conditional
    pp-line
    pp-diagnostic
    pp-region

pp-new-line::
    whitespace$_{opt}$ *single-line-comment*$_{opt}$ *new-line*

conditional-symbol::
    Any *identifier-or-keyword* except `true` or `false`

pp-expression::
    whitespace$_{opt}$ *pp-or-expression* *whitespace*$_{opt}$

pp-or-expression::
    pp-and-expression
    pp-or-expression *whitespace*$_{opt}$ `||` *whitespace*$_{opt}$ *pp-and-expression*

pp-and-expression::
    pp-equality-expression
    pp-and-expression *whitespace*$_{opt}$ && *whitespace*$_{opt}$ *pp-equality-expression*

pp-equality-expression::
    pp-unary-expression
    pp-equality-expression *whitespace*$_{opt}$ `==` *whitespace*$_{opt}$ *pp-unary-expression*
    pp-equality-expression *whitespace*$_{opt}$ `!=` *whitespace*$_{opt}$ *pp-unary-expression*

pp-unary-expression::
    pp-primary-expression
    `!` *whitespace*$_{opt}$ *pp-unary-expression*

pp-primary-expression::
    `true`
    `false`
    conditional-symbol
    `(` *whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$ `)`

pp-declaration::
    whitespace$_{opt}$ # *whitespace*$_{opt}$ `define` *whitespace* *conditional-symbol* *pp-new-line*
    whitespace$_{opt}$ # *whitespace*$_{opt}$ `undef` *whitespace* *conditional-symbol* *pp-new-line*

pp-conditional::
    pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*

pp-if-section::
    whitespace$_{opt}$ # *whitespace*$_{opt}$ `if` *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-elif-sections::
    pp-elif-section
    pp-elif-sections *pp-elif-section*

pp-elif-section::
    whitespace$_{opt}$ # *whitespace*$_{opt}$ `elif` *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-else-section::
    whitespace$_{opt}$ # *whitespace*$_{opt}$ `else` *pp-new-line* *conditional-section*$_{opt}$

pp-endif::
  whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endif` *pp-new-line*

conditional-section::
  input-section
  skipped-section

skipped-section::
  skipped-section-part
  skipped-section skipped-section-part

skipped-section-part::
  skipped-characters$_{opt}$ *new-line*
  pp-directive

skipped-characters::
  whitespace$_{opt}$ *not-number-sign input-characters*$_{opt}$

not-number-sign::
  Any *input-character* except `#`

pp-line::
  whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `line` *whitespace*$_{opt}$ *line-indicator pp-new-line*

line-indicator::
  decimal-digits whitespace file-name
  decimal-digits
  `default`

file-name::
  `"` *file-name-characters* `"`

file-name-characters::
  file-name-character
  file-name-characters file-name-character

file-name-character::
  Any character except `"` (`U+0022`), and *new-line*

pp-diagnostic::
  whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `error` *whitespace*$_{opt}$ *pp-message*
  whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `warning` *whitespace*$_{opt}$ *pp-message*

pp-message::
  input-characters$_{opt}$ *new-line*

pp-region::
  pp-start-region conditional-section$_{opt}$ *pp-end-region*

pp-start-region::
  whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `region` *whitespace*$_{opt}$ *pp-message*

pp-end-region::
  whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endregion` *whitespace*$_{opt}$ *pp-message*

A.2 Syntactic grammar

A.2.1 Basic concepts

namespace-name:
  namespace-or-type-name

type-name:
  namespace-or-type-name

namespace-or-type-name:
identifier
namespace-or-type-name `.` *identifier*

A.2.2 Types

type:
value-type
reference-type

value-type:
struct-type
enum-type

struct-type:
type-name
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`
`char`

floating-point-type:
`float`
`double`

enum-type:
type-name

reference-type:
class-type
interface-type
array-type
delegate-type

class-type:
type-name
`object`
`string`

interface-type:
type-name

array-type:
non-array-type rank-specifiers

non-array-type:
  type

rank-specifiers:
  rank-specifier
  rank-specifiers rank-specifier

rank-specifier:
  `[` *dim-separators*$_{opt}$ `]`

dim-separators:
  `,`
  dim-separators `,`

delegate-type:
  type-name

A.2.3 Variables

variable-reference:
  expression

A.2.4 Expressions

argument-list:
  argument
  argument-list `,` *argument*

argument:
  expression
  `ref` *variable-reference*
  `out` *variable-reference*

primary-expression:
  array-creation-expression
  primary-no-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

  typeof-expression
  sizeof-expression
  checked-expression
  unchecked-expression

simple-name:
  identifier

parenthesized-expression:
  `(` *expression* `)`

member-access:
  primary-expression `.` *identifier*
  predefined-type `.` *identifier*

predefined-type: one of
  `bool` `byte` `char` `decimal` `double` `float` `int` `long`
  `object` `sbyte` `short` `string` `uint` `ulong` `ushort`

invocation-expression:
  primary-expression `(` *argument-list*$_{opt}$ `)`

element-access:
  primary-no-array-creation-expression `[` *expression-list* `]`

expression-list:
  expression
  expression-list `,` *expression*

this-access:
  `this`

base-access:
  `base` `.` *identifier*
  `base` `[` *expression-list* `]`

post-increment-expression:
  primary-expression `++`

post-decrement-expression:
  primary-expression `--`

object-creation-expression:
  `new` *type* `(` *argument-list*$_{opt}$ `)`

array-creation-expression:
  `new` *non-array-type* `[` *expression-list* `]` *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
  `new` *array-type* *array-initializer*

delegate-creation-expression:
  `new` *delegate-type* `(` *expression* `)`

typeof-expression:
  `typeof` `(` *type* `)`
  `typeof` `(` `void` `)`

checked-expression:
  `checked` `(` *expression* `)`

unchecked-expression:
  `unchecked` `(` *expression* `)`

unary-expression:
  primary-expression
  `+` *unary-expression*
  `-` *unary-expression*
  `!` *unary-expression*
  `~` *unary-expression*
  `*` *unary-expression*
  pre-increment-expression
  pre-decrement-expression
  cast-expression

pre-increment-expression:
  `++` *unary-expression*

pre-decrement-expression:
-- unary-expression

cast-expression:
(type) unary-expression

multiplicative-expression:
unary-expression
multiplicative-expression * unary-expression
multiplicative-expression / unary-expression
multiplicative-expression % unary-expression

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression – multiplicative-expression

shift-expression:
additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression

relational-expression:
shift-expression
relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
relational-expression is type
relational-expression as type

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

and-expression:
equality-expression
and-expression & equality-expression

exclusive-or-expression:
and-expression
exclusive-or-expression ^ and-expression

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | exclusive-or-expression

conditional-and-expression:
inclusive-or-expression
conditional-and-expression && inclusive-or-expression

conditional-or-expression:
conditional-and-expression
conditional-or-expression || conditional-and-expression

conditional-expression:
conditional-or-expression
conditional-or-expression ? expression : expression

assignment:
unary-expression assignment-operator expression

assignment-operator: *one of*
`=  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=`

expression:
conditional-expression
assignment

constant-expression:
expression

boolean-expression:
expression

A.2.5 Statements

statement:
labeled-statement
declaration-statement
embedded-statement

embedded-statement:
block
empty-statement
expression-statement
selection-statement
iteration-statement
jump-statement
try-statement
checked-statement
unchecked-statement
lock-statement
using-statement

block:
`{` *statement-list*$_{opt}$ `}`

statement-list:
statement
statement-list *statement*

empty-statement:
`;`

labeled-statement:
identifier `:` *statement*

declaration-statement:
local-variable-declaration `;`
local-constant-declaration `;`

local-variable-declaration:
type *local-variable-declarators*

local-variable-declarators:
local-variable-declarator
local-variable-declarators `,` *local-variable-declarator*

local-variable-declarator:
identifier
identifier `=` *local-variable-initializer*

local-variable-initializer:
expression
array-initializer

local-constant-declaration:
`const` *type* *constant-declarators*

constant-declarators:
constant-declarator
constant-declarators `,` *constant-declarator*

constant-declarator:
identifier `=` *constant-expression*

expression-statement:
statement-expression `;`

statement-expression:
invocation-expression
object-creation-expression
assignment
post-increment-expression
post-decrement-expression
pre-increment-expression
pre-decrement-expression

selection-statement:
if-statement
switch-statement

if-statement:
`if` `(` *boolean-expression* `)` *embedded-statement*
`if` `(` *boolean-expression* `)` *embedded-statement* `else` *embedded-statement*

boolean-expression:
expression

switch-statement:
`switch` `(` *expression* `)` *switch-block*

switch-block:
`{` *switch-sections*$_{opt}$ `}`

switch-sections:
switch-section
switch-sections *switch-section*

switch-section:
switch-labels *statement-list*

switch-labels:
switch-label
switch-labels *switch-label*

switch-label:
`case` *constant-expression* `:`
`default` `:`

iteration-statement:
while-statement
do-statement
for-statement
foreach-statement

while-statement:
    `while ( ` *boolean-expression* ` ) ` *embedded-statement*

do-statement:
    `do` *embedded-statement* `while ( ` *boolean-expression* ` ) ;`

for-statement:
    `for ( ` *for-initializer*$_{opt}$ ` ; ` *for-condition*$_{opt}$ ` ; ` *for-iterator*$_{opt}$ ` ) ` *embedded-statement*

for-initializer:
    local-variable-declaration
    statement-expression-list

for-condition:
    boolean-expression

for-iterator:
    statement-expression-list

statement-expression-list:
    statement-expression
    statement-expression-list ` , ` *statement-expression*

foreach-statement:
    `foreach ( ` *type identifier* ` in ` *expression* ` ) ` *embedded-statement*

jump-statement:
    break-statement
    continue-statement
    goto-statement
    return-statement
    throw-statement

break-statement:
    `break ;`

continue-statement:
    `continue ;`

goto-statement:
    `goto` *identifier* `;`
    `goto case` *constant-expression* *;*
    `goto default ;`

return-statement:
    `return` *expression*$_{opt}$ `;`

throw-statement:
    `throw` *expression*$_{opt}$ `;`

try-statement:
    `try` *block catch-clauses*
    `try` *block finally-clause*
    `try` *block catch-clauses finally-clause*

catch-clauses:
    specific-catch-clauses general-catch-clause$_{opt}$
    specific-catch-clauses$_{opt}$ *general-catch-clause*

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause

specific-catch-clause:
  `catch` `(` *class-type* *identifier*$_{opt}$ `)` *block*

general-catch-clause:
  `catch` *block*

finally-clause:
  `finally` *block*

checked-statement:
  `checked` *block*

unchecked-statement:
  `unchecked` *block*

lock-statement:
  `lock` `(` *expression* `)` *embedded-statement*

using-statement:
  `using` `(` *resource-acquisition* `)` *embedded-statement*

resource-acquisition:
  local-variable-declaration
  expression

compilation-unit:
  using-directives$_{opt}$ *global-attributes*$_{opt}$ *namespace-member-declarations*$_{opt}$

namespace-declaration:
  `namespace` *qualified-identifier* *namespace-body* `;`$_{opt}$

qualified-identifier:
  identifier
  qualified-identifier `.` *identifier*

namespace-body:
  `{` *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ `}`

using-directives:
  using-directive
  using-directives *using-directive*

using-directive:
  using-alias-directive
  using-namespace-directive

using-alias-directive:
  `using` *identifier* `=` *namespace-or-type-name* `;`

using-namespace-directive:
  `using` *namespace-name* `;`

namespace-member-declarations:
  namespace-member-declaration
  namespace-member-declarations *namespace-member-declaration*

namespace-member-declaration:
  namespace-declaration
  type-declaration

type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration

A.2.6 Classes

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `class` *identifier* *class-base*$_{opt}$ *class-body* `;`$_{opt}$

class-modifiers:
class-modifier
class-modifiers *class-modifier*

class-modifier:
`new`
`public`
`protected`
`internal`
`private`
`abstract`
`sealed`

class-base:
`:` *class-type*
`:` *interface-type-list*
`:` *class-type* `,` *interface-type-list*

interface-type-list:
interface-type
interface-type-list `,` *interface-type*

class-body:
`{` *class-member-declarations*$_{opt}$ `}`

class-member-declarations:
class-member-declaration
class-member-declarations *class-member-declaration*

class-member-declaration:
constant-declaration
field-declaration
method-declaration
property-declaration
event-declaration
indexer-declaration
operator-declaration
constructor-declaration
destructor-declaration
static-constructor-declaration
type-declaration

constant-declaration:
attributes$_{opt}$ *constant-modifiers*$_{opt}$ `const` *type* *constant-declarators* `;`

constant-modifiers:
constant-modifier
constant-modifiers *constant-modifier*

constant-modifier:
`new`
`public`
`protected`
`internal`
`private`

constant-declarators:
constant-declarator
constant-declarators `,` *constant-declarator*

constant-declarator:
identifier `=` *constant-expression*

field-declaration:
$attributes_{opt}$ $field\text{-}modifiers_{opt}$ *type* *variable-declarators* `;`

field-modifiers:
field-modifier
field-modifiers *field-modifier*

field-modifier:
`new`
`public`
`protected`
`internal`
`private`
`static`
`readonly`
`volatile`

variable-declarators:
variable-declarator
variable-declarators `,` *variable-declarator*

variable-declarator:
identifier
identifier `=` *variable-initializer*

variable-initializer:
expression
array-initializer

method-declaration:
method-header *method-body*

method-header:
$attributes_{opt}$ $method\text{-}modifiers_{opt}$ *return-type* *member-name* `(` $formal\text{-}parameter\text{-}list_{opt}$ `)`

method-modifiers:
method-modifier
method-modifiers *method-modifier*

method-modifier:
`new`
`public`
`protected`
`internal`
`private`
`static`
`virtual`
`sealed`
`override`
`abstract`
`extern`

return-type:
type
`void`

member-name:
identifier
interface-type `.` *identifier*

method-body:
block
`;`

formal-parameter-list:
fixed-parameters
fixed-parameters `,` *parameter-array*
parameter-array

fixed-parameters:
fixed-parameter
fixed-parameters `,` *fixed-parameter*

fixed-parameter:
$attributes_{opt}$ $parameter\text{-}modifier_{opt}$ *type* *identifier*

parameter-modifier:
`ref`
`out`

parameter-array:
$attributes_{opt}$ `params` *array-type* *identifier*

property-declaration:
$attributes_{opt}$ $property\text{-}modifiers_{opt}$ *type* *member-name* `{` *accessor-declarations* `}`

property-modifiers:
property-modifier
property-modifiers *property-modifier*

property-modifier:
`new`
`public`
`protected`
`internal`
`private`
`static`
`virtual`
`sealed`
`override`
`abstract`
`extern`

member-name:
identifier
interface-type `.` *identifier*

accessor-declarations:
get-accessor-declaration *set-accessor-declaration*$_{opt}$
set-accessor-declaration *get-accessor-declaration*$_{opt}$

get-accessor-declaration:
attributes$_{opt}$ `get` *accessor-body*

set-accessor-declaration:
attributes$_{opt}$ `set` *accessor-body*

accessor-body:
block
`;`

event-declaration:
attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *variable-declarators* `;`
attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type* *member-name* `{` *event-accessor-declarations* `}`

event-modifiers:
event-modifier
event-modifiers *event-modifier*

event-modifier:
`new`
`public`
`protected`
`internal`
`private`
`static`
`virtual`
`sealed`
`override`
`abstract`
`extern`

event-accessor-declarations:
add-accessor-declaration *remove-accessor-declaration*
remove-accessor-declaration *add-accessor-declaration*

add-accessor-declaration:
attributes$_{opt}$ `add` *block*

remove-accessor-declaration:
  attributes$_{opt}$ `remove` *block*

indexer-declaration:
  attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* `{` *accessor-declarations* `}`

indexer-modifiers:
  indexer-modifier
  indexer-modifiers *indexer-modifier*

indexer-modifier:
  `new`
  `public`
  `protected`
  `internal`
  `private`
  `virtual`
  `sealed`
  `override`
  `abstract`
  `extern`

indexer-declarator:
  type `this` `[` *formal-parameter-list* `]`
  type *interface-type* `.` `this` `[` *formal-parameter-list* `]`

operator-declaration:
  attributes$_{opt}$ *operator-modifiers* *operator-declarator* *operator-body*

operator-modifiers:
  operator-modifier
  operator-modifiers *operator-modifier*

operator-modifier:
  `public`
  `static`
  `extern`

operator-declarator:
  unary-operator-declarator
  binary-operator-declarator
  conversion-operator-declarator

unary-operator-declarator:
  type `operator` *overloadable-unary-operator* `(` *type* *identifier* `)`

overloadable-unary-operator: one of
  `+` `-` `!` `~` `++` `--` `true` `false`

binary-operator-declarator:
  type `operator` *overloadable-binary-operator* `(` *type* *identifier* `,` *type* *identifier* `)`

overloadable-binary-operator: one of
  `+` `-` `*` `/` `%` `&` `|` `^` `<<` `>>` `==` `!=` `>` `<` `>=` `<=`

conversion-operator-declarator:
  `implicit` `operator` *type* `(` *type* *identifier* `)`
  `explicit` `operator` *type* `(` *type* *identifier* `)`

operator-body:
  block
  `;`

constructor-declaration:
attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator* *constructor-body*

constructor-modifiers:
constructor-modifier
constructor-modifiers *constructor-modifier*

constructor-modifier:
`public`
`protected`
`internal`
`private`
`extern`

constructor-declarator:
identifier `(` *formal-parameter-list*$_{opt}$ `)` *constructor-initializer*$_{opt}$

constructor-initializer:
`:` `base` `(` *argument-list*$_{opt}$ `)`
`:` `this` `(` *argument-list*$_{opt}$ `)`

constructor-body:
block
`;`

static-constructor-declaration:
attributes$_{opt}$ *static-constructor-modifiers* *identifier* `(` `)` *static-constructor-body*

static-constructor-modifiers:
`extern`$_{opt}$ `static`
`static` `extern`$_{opt}$

static-constructor-body:
block
`;`

destructor-declaration:
attributes$_{opt}$ `extern`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*

destructor-body:
block
`;`

A.2.7 Structs

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ `struct` *identifier* *struct-interfaces*$_{opt}$ *struct-body* `;`$_{opt}$

struct-modifiers:
struct-modifier
struct-modifiers *struct-modifier*

struct-modifier:
`new`
`public`
`protected`
`internal`
`private`

struct-interfaces:
`:` *interface-type-list*

struct-body:
    `{` *struct-member-declarations*$_{opt}$ `}`

struct-member-declarations:
    struct-member-declaration
    struct-member-declarations struct-member-declaration

struct-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration

A.2.8 Arrays

array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    `[` *dim-separators*$_{opt}$ `]`

dim-separators:
    `,`
    dim-separators `,`

array-initializer:
    `{` *variable-initializer-list*$_{opt}$ `}`
    `{` *variable-initializer-list* `,` `}`

variable-initializer-list:
    variable-initializer
    variable-initializer-list `,` *variable-initializer*

variable-initializer:
    expression
    array-initializer

A.2.9 Interfaces

interface-declaration:
    attributes$_{opt}$ *interface-modifiers*$_{opt}$ `interface` *identifier* *interface-base*$_{opt}$ *interface-body* `;`$_{opt}$

interface-modifiers:
    interface-modifier
    interface-modifiers interface-modifier

interface-modifier:
`new`
`public`
`protected`
`internal`
`private`

interface-base:
`:` *interface-type-list*

interface-body:
`{` *interface-member-declarations*$_{opt}$ `}`

interface-member-declarations:
interface-member-declaration
interface-member-declarations *interface-member-declaration*

interface-member-declaration:
interface-method-declaration
interface-property-declaration
interface-event-declaration
interface-indexer-declaration

interface-method-declaration:
attributes$_{opt}$ `new`$_{opt}$ *return-type* *identifier* `(` *formal-parameter-list*$_{opt}$ `)` `;`

interface-property-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* *identifier* `{` *interface-accessors* `}`

interface-accessors:
attributes$_{opt}$ `get` `;`
attributes$_{opt}$ `set` `;`
attributes$_{opt}$ `get` `;` *attributes*$_{opt}$ `set` `;`
attributes$_{opt}$ `set` `;` *attributes*$_{opt}$ `get` `;`

interface-event-declaration:
attributes$_{opt}$ `new`$_{opt}$ `event` *type* *identifier* `;`

interface-indexer-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* `this` `[` *formal-parameter-list* `]` `{` *interface-accessors* `}`

A.2.10 Enums

enum-declaration:
attributes$_{opt}$ *enum-modifiers*$_{opt}$ `enum` *identifier* *enum-base*$_{opt}$ *enum-body* `;`$_{opt}$

enum-base:
`:` *integral-type*

enum-body:
`{` *enum-member-declarations*$_{opt}$ `}`
`{` *enum-member-declarations* `,` `}`

enum-modifiers:
enum-modifier
enum-modifiers *enum-modifier*

enum-modifier:
`new`
`public`
`protected`
`internal`
`private`

enum-member-declarations:
    enum-member-declaration
    enum-member-declarations `,` *enum-member-declaration*

enum-member-declaration:
    attributes$_{opt}$ *identifier*
    attributes$_{opt}$ *identifier* `=` *constant-expression*

A.2.11 Delegates

delegate-declaration:
    attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *type* *identifier* `(` *formal-parameter-list*$_{opt}$ `)`
    `;`

delegate-modifiers:
    delegate-modifier
    delegate-modifiers *delegate-modifier*

delegate-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

A.2.12 Attributes

global-attributes:
    global-attribute-sections

global-attribute-sections:
    global-attribute-section
    global-attribute-sections *global-attribute-section*

global-attribute-section:
    `[` *global-attribute-target-specifier* *attribute-list* `]`
    `[` *global-attribute-target-specifier* *attribute-list* `,]`

global-attribute-target-specifier:
    global-attribute-target `:`

global-attribute-target:
    `assembly`

attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections *attribute-section*

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `,]`

attribute-target-specifier:
    attribute-target `:`

attribute-target:
`field`
`event`
`method`
`module`
`param`
`property`
`return`
`type`

attribute-list:
attribute
attribute-list `,` *attribute*

attribute:
attribute-name *attribute-arguments*$_{opt}$

attribute-name:
type-name

attribute-arguments:
`(` *positional-argument-list*$_{opt}$ `)`
`(` *positional-argument-list* `,` *named-argument-list* `)`
`(` *named-argument-list* `)`

positional-argument-list:
positional-argument
positional-argument-list `,` *positional-argument*

positional-argument:
attribute-argument-expression

named-argument-list:
named-argument
named-argument-list `,` *named-argument*

named-argument:
identifier `=` *attribute-argument-expression*

attribute-argument-expression:
expression

A.3 Grammar extensions for unsafe code

embedded-statement:
...
unsafe-statement

unsafe-statement:
`unsafe` *block*

type:
value-type
reference-type
pointer-type

pointer-type:
unmanaged-type `*`
`void` `*`

unmanaged-type:
type

primary-no-array-creation-expression:
...
pointer-member-access
pointer-element-access
sizeof-expression

unary-expression:
...
pointer-indirection-expression
addressof-expression

pointer-indirection-expression:
`*` *unary-expression*

pointer-member-access:
primary-expression `->` *identifier*

pointer-element-access:
primary-no-array-creation-expression `[` *expression* `]`

addressof-expression:
`&` *unary-expression*

sizeof-expression:
`sizeof` `(` *unmanaged-type* `)`

embedded-statement:
...
fixed-statement

fixed-statement:
`fixed` `(` *pointer-type* *fixed-pointer-declarators* `)` *embedded-statement*

fixed-pointer-declarators:
fixed-pointer-declarator
fixed-pointer-declarators `,` *fixed-pointer-declarator*

fixed-pointer-declarator:
identifier `=` *fixed-pointer-initializer*

fixed-pointer-initializer:
`&` *variable-reference*
expression

variable-initializer:
expression
array-initializer
stackalloc-initializer

stackalloc-initializer:
`stackalloc` *unmanaged-type* `[` *expression* `]`

End of informative text.

B. Portability issues

This clause is informative.

This annex collects some information about portability that appears in this ECMA Standard.

B.1 Undefined behavior

A program that does not contain any occurrences of the `unsafe` modifier cannot exhibit any undefined behavior.

The behavior is undefined in the following circumstances:

1. When dereferencing the result of converting one pointer type to another, and the resulting pointer is not correctly aligned for the pointed-to type. (§25.4)
2. When the unary `*` operator is applied to a pointer containing an invalid value (§25.5.1).
3. When a pointer is subscripted to access an out-of-bounds element (§25.5.3).
4. Modifying objects of managed type through fixed pointers (§25.6)
5. The initial content of memory allocated by `stackalloc` (§25.7).

B.2 Implementation-defined behavior

A conforming implementation is required to document its choice of behavior in each of the areas listed in this clause. The following are implementation-defined:

1. The behavior when an identifier not in Normalization Form C is encountered (§9.4.2).
2. The values of any application parameters passed to `Main` by the host environment prior to application startup (§10.1).
3. The mechanism by which linkage to an external function is achieved (§17.5.7).
4. The impact of thread termination when no matching catch clause is found for an exception and the code that initially started that thread is reached. (§23.3)
5. The purpose of attribute target specifiers other than those identified by this standard (§24.2).
6. The mappings between pointers and integers (§25.4).
7. The effect of applying the unary `*` operator to a `null` pointer (§25.5.1).
8. The behavior when pointer arithmetic overflows the domain of the pointer type (§25.5.5).
9. The result of the `sizeof` operator for other than the pre-defined value types (§25.5.8).
10. The behavior of the `fixed` statement if the array expression is null or if the array has zero elements (§25.6).

11. The behavior of the `fixed` statement if the string expression is null (§25.6).
12. The value returned when a stack allocation of size zero is made (§25.7).

B.3 Unspecified behavior

1. The time at which the destructor (if any) for an object is run, once that object has become eligible for destruction.
2. The value of the result when converting out-of-range values from `float`or `double` values to an integral type in an unchecked context (§13.2.1).
3. The layout of arrays, except in an unsafe context (§14.5.10.2).
4. The exact timing of static field initialization (§17.4.5.1).
5. The order in which members are packed into a struct (§25.5.8).

B.4 Other Issues

1. The exact results of floating-point expression evaluation may vary from one implementation to another, because an implementation is permitted to evaluate such expressions using a greater range and/or precision than is required. (§11.1.5)
2. The CLI reserves certain signatures for compatibility with other programming languages. (§17.2.7)

End of informative text.

C. Naming guidelines

This annex is informative.

One of the most important elements of predictability and discoverability is the use of a consistent naming pattern. Many of the common user questions don't even arise once these conventions are understood and widely used. There are three elements to the naming guidelines:

1. Casing – use of the correct capitalization style
2. Mechanical – use nouns for classes, verbs for methods, etc.
3. Word choice – use consistent terms across class libraries.

The following section lays out rules for the first two elements, and some philosophy for the third.

C.1 Capitalization styles

The following section describes different ways of capitalizing identifiers.

C.1.1 Pascal casing

This convention capitalizes the first character of each word. For example:

```
Color    BitConverter
```

C.1.2 Camel casing

This convention capitalizes the first character of each word except the first word. For example:

```
backgroundColor    totalValueCount
```

C.1.3 All uppercase

Only use all uppercase letters for an identifier if it contains an abbreviation. For example:

```
System.IO
System.WinForms.UI
```

C.1.4 Capitalization summary

The following table summarizes the capitalization style for the different kinds of identifiers:

Type	Case	Notes
Class	PascalCase	
Attribute Class	PascalCase	Has a suffix of `Attribute`
Exception Class	PascalCase	Has a suffix of `Exception`
Constant	PascalCase	
Enum type	PascalCase	
Enum values	PascalCase	
Event	PascalCase	
Interface	PascalCase	Has a prefix of `I`
Local variable	camelCase	
Method	PascalCase	
Namespace	PascalCase	
Property	PascalCase	
Public Instance Field	PascalCase	Rarely used (use a property instead)
Protected Instance Field	camelCase	Rarely used (use a property instead)
Parameter	camelCase	

C.2 Word choice

- Do avoid using class names duplicated in heavily used namespaces. For example, don't use the following for a class name.

```
System    Collections    Forms    UI
```

- Do not use abbreviations in identifiers.
- If you must use abbreviations, do use camelCase for any abbreviation containing more than two characters, even if this is not the usual abbreviation.

C.3 Namespaces

The general rule for namespace naming is: `CompanyName.TechnologyName`.

- Do avoid the possibility of two published namespaces having the same name, by prefixing namespace names with a company name or other well-established brand. For example, `Microsoft.Office` for the Office Automation classes provided by Microsoft.
- Do use PascalCase, and separate logical components with periods (as in `Microsoft.Office.PowerPoint`). If your brand employs non-traditional casing, do follow the casing defined by your brand, even if it deviates from normal namespace casing (for example, `NeXT.WebObjects`, and `ee.cummings`).
- Do use plural namespace names where appropriate. For example, use `System.Collections` rather than `System.Collection`. Exceptions to this rule are brand names and abbreviations. For example, use `System.IO` not `System.IOs`.
- Do not have namespaces and classes with the same name.

C.4 Classes

- Do name classes with nouns or noun phrases.

- Do use PascalCase.
- Do use sparingly, abbreviations in class names.
- Do not use any prefix (such as "C", for example). Where possible, avoid starting with the letter "I", since that is the recommended prefix for interface names. If you must start with that letter, make sure the second character is lowercase, as in `IdentityStore`.
- Do not use any underscores.

```
public class FileStream { … }
public class Button { … }
public class String { … }
```

C.5 Interfaces

- Do name interfaces with nouns or noun phrases, or adjectives describing behavior. For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective).
- Do use PascalCase.
- Do use sparingly, abbreviations in interface names.
- Do not use any underscores.
- Do prefix interface names with the letter "I", to indicate that the type is an interface.
- Do use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the "I" prefix in the interface name. This approach is used for the interface `IComponent` and its standard implementation, Component.

```
public interface IComponent { … }
public class Component : IComponent { … }
public interface IServiceProvider{ … }
public interface IFormatable { … }
```

C.6 Enums

- Do use PascalCase for enums.
- Do use PascalCase for enum value names.
- Do use sparingly, abbreviations in enum names.
- Do not use a family-name prefix on enum.
- Do not use any "Enum" suffix on enum types.
- Do use a singular name for enums
- Do use a plural name for bit fields
- Do define enumerated values using an enum if they are used in a parameter or property. This gives development tools a chance at knowing the possible values for a property or parameter.

```
public enum FileMode{
    Create,
    CreateNew,
    Open,
    OpenOrCreate,
    Truncate
}
```

- Do use the Flags custom attribute if the numeric values are meant to be bitwise ORed together

```
[Flags]
public enum Bindings {
    CreateInstance,
    DefaultBinding,
    ExcatBinding,
    GetField,
    GetProperty,
    IgnoreCase,
    InvokeMethod,
    NonPublic,
    OABinding,
    SetField
    SetProperty,
    Static
}
```

- Do use `int` as the underlying type of an enum. (An exception to this rule is if the enum represents flags and there are more than 32 flags, or the enum may grow to that many flags in the future, or the type needs to be different from `int` for backward compatibility.)
- Do use enums only if the value can be completely expressed as a set of bit flags. Do not use enums for open sets (such as operating system version).

C.7 Static fields

- Do name static members with nouns, noun phrases, or abbreviations for nouns.
- Do name static members using PascalCase.
- Do not use Hungarian-type prefixes on static member names.

C.8 Parameters

- Do use descriptive names such that a parameter's name and type clearly imply its meaning.
- Do name parameters using camelCase.
- Do prefer names based on a parameter's meaning, to names based on the parameter's type. It is likely that development tools will provide the information about type in a convenient way, so the parameter name can be put to better use describing semantics rather than type.
- Do not reserve parameters for future use. If more data is need in the next version, a new overload can be added.
- Do not use Hungarian-type prefixes.

```
Type GetType (string typeName)
string Format (string format, object [] args)
```

C.9 Methods

- Do name methods with verbs or verb phrases.
- Do name methods with PascalCase

```
RemoveAll(), GetCharArray(), Invoke()
```

C.10 Properties

- Do name properties using noun or noun phrases
- Do name properties with PascalCase
- Consider having a property with the same as a type. When declaring a property with the same name as a type, also make the type of the property be that type. In other words, the following is okay

```
public enum Color {...}
public class Control {
    public Color Color { get {...} set {...} }
}
```

but this is not

```
public enum Color {...}
public class Control {
    public int Color { get {...} set {...} }
}
```

In the latter case, it will not be possible to refer to the members of the Color enum because `Color.Xxx` will be interpreted as being a member access that first gets the value of the `Color` property (of type `int`) and then accesses a member of that value (which would have to be an instance member of `System.Int32`).

C.11 Events

- <u>Do</u> name event handlers with the "EventHandler" suffix.

```
public delegate void MouseEventHandler(object sender, MouseEvent e);
```

- <u>Do</u> use two parameters named `sender` and `e`. The sender parameter represents the object that raised the event, and this parameter is always of type `object`, even if it is possible to employ a more specific type. The state associated with the event is encapsulated in an instance `e` of an event class. Use an appropriate and specific event class for its type.

```
public delegate void MouseEventHandler(object sender, MouseEvent e);
```

- <u>Do</u> name event argument classes with the "EventArgs" suffix.

```
public class MouseEventArgs : EventArgs {
    int x;
    int y;
    public MouseEventArgs(int x, int y)
       { this.x = x; this.y = y; }
    public int X { get { return x; } }
    public int Y { get { return y; } }
}
```

- <u>Do</u> name event names that have a concept of pre- and post-operation using the present and past tense (do not use `BeforeXxx`/`AfterXxx` pattern). For example, a close event that could be canceled would have a Closing and Closed event.

```
public event ControlEventHandler ControlAdded {
  //..
}
```

- <u>Consider</u> naming events with a verb.

C.12 Case sensitivity

- <u>Don't</u> use names that require case sensitivity. Components might need to be usable from both case-sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between two names within the same context that differ only by case, components must avoid this situation.

Examples of what not to do:

- <u>Don't</u> have two namespaces whose names differ only by case.

```
namespace ee.cummings;
namespace Ee.Cummings;
```

- <u>Don't</u> have a method with two parameters whose names differ only by case.

```
void F(string a, string A)
```

- Don't have a namespace with two types whose names differ only by case.

```
System.WinForms.Point p;
System.WinForms.POINT pp;
```

- Don't have a type with two properties whose names differ only by case.

```
int F {get, set};
int F {get, set}
```

- Don't have a type with two methods whose names differ only by case.

```
void f();
void F();
```

C.13 Avoiding type name confusion

Different languages use different names to identify the fundamental managed types, so in a multi-language environment, designers must take care to avoid language-specific terminology. This section describes a set of rules that help avoid type name confusion.

- Do use semantically interesting names rather than type names.
- In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have:

```
void Write(double value);
void Write(float value);
void Write(long value);
void Write(int value);
void Write(short value);
```

rather than a language-specific alternative such as:

```
void Write(double doubleValue);
void Write(float floatValue);
void Write(long longValue);
void Write(int intValue);
void Write(short shortValue);
```

- In the extremely rare case that it is necessary to have a uniquely named method for each fundamental data type, do use the following universal type names: `Sbyte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Boolean`, `Char`, `String`, and `Object`. For example, a class that supports reading a variety of data types from a stream might have:

```
double ReadDouble();
float ReadSingle();
long ReadIn64();
int ReadInt32();
short ReadInt16();
```

rather than a language-specific alternative such as:

```
double ReadDouble();
float ReadFloat();
long ReadLong();
int ReadInt();
short ReadShort();
```

End of informative text.

D. Standard Library

A conforming C# implementation shall provide a set of types having specific semantics. For convenience, these types and their members are listed here, in alphabetical order. For a formal definition of these types and their members, refer to ECMA-xxx, 1st Edition, December 2001, *Common Language Infrastructure (CLI)*, *Partition IV; Base Class Library (BCL)*, *Extended Numerics Library*, *and Extended Array Library*, which are included by reference in this ECMA Standard.

This rest of this clause is informative.

```
// Namespace: System, Library: BCL
public class ApplicationException: Exception
{
   public ApplicationException();
   public ApplicationException(string message);
   public ApplicationException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public class ArgumentException: SystemException
{
   public ArgumentException();
   public ArgumentException(string message);
   public ArgumentException(string message, Exception innerException);
   public ArgumentException(string message, string paramName, Exception
      innerException);
   public ArgumentException(string message, string paramName);
   public virtual string ParamName { get; }
}

// Namespace: System, Library: BCL
public class ArgumentNullException: ArgumentException
{
   public ArgumentNullException();
   public ArgumentNullException(string paramName);
   public ArgumentNullException(string paramName, string message);
}

// Namespace: System, Library: BCL
public class ArgumentOutOfRangeException: ArgumentException
{
   public ArgumentOutOfRangeException();
```

```
   public ArgumentOutOfRangeException(string paramName);
   public ArgumentOutOfRangeException(string paramName, string message);
   public ArgumentOutOfRangeException(string paramName, object actualValue,
      string message);
   public virtual object ActualValue { get; }
}

// Namespace: System, Library: BCL
public class ArithmeticException: SystemException
{
   public ArithmeticException();
   public ArithmeticException(string message);
   public ArithmeticException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public abstract class Array: ICloneable, ICollection, IEnumerable, IList
{
   protected Array();
   public static int BinarySearch(Array array, object value);
   public static int BinarySearch(Array array, int index, int length, object
      value);
   public static int BinarySearch(Array array, object value, IComparer
      comparer);
   public static int BinarySearch(Array array, int index, int length, object
      value, IComparer comparer);
   public static void Clear(Array array, int index, int length);
   public virtual object Clone();
   public static void Copy(Array sourceArray, Array destinationArray, int
      length);
   public static void Copy(Array sourceArray, int sourceIndex, Array
      destinationArray, int destinationIndex, int length);
   public virtual void CopyTo(Array array, int index);
   public static Array CreateInstance(Type elementType, int length);
   public static Array CreateInstance(Type elementType, int length1, int
      length2);
   public static Array CreateInstance(Type elementType, int length1, int
      length2, int length3);
   public static Array CreateInstance(Type elementType, int[] lengths);
   public static Array CreateInstance(Type elementType, int[] lengths, int[]
      lowerBounds);
   public virtual IEnumerator GetEnumerator();
   public object GetValue(int[] indices);
   public object GetValue(int index);
```

```
    public object GetValue(int index1, int index2);
    public object GetValue(int index1, int index2, int index3);
    public static int IndexOf(Array array, object value);
    public static int IndexOf(Array array, object value, int startIndex);
    public static int IndexOf(Array array, object value, int startIndex, int
       count);
    public static int LastIndexOf(Array array, object value);
    public static int LastIndexOf(Array array, object value, int startIndex);
    public static int LastIndexOf(Array array, object value, int startIndex,
       int count);
    public static void Reverse(Array array);
    public static void Reverse(Array array, int index, int length);
    public void SetValue(object value, int index);
    public void SetValue(object value, int index1, int index2);
    public void SetValue(object value, int index1, int index2, int index3);
    public void SetValue(object value, int[] indices);
    public static void Sort(Array array);
    public static void Sort(Array keys, Array items);
    public static void Sort(Array array, int index, int length);
    public static void Sort(Array keys, Array items, int index, int length);
    public static void Sort(Array array, IComparer comparer);
    public static void Sort(Array keys, Array items, IComparer comparer);
    public static void Sort(Array array, int index, int length, IComparer
       comparer);
    public static void Sort(Array keys, Array items, int index, int length,
       IComparer comparer);
    int IList.Add(object value);
    void IList.Clear();
    bool IList.Contains(object value);
    int IList.IndexOf(object value);
    void IList.Insert(int index, object value);
    void IList.Remove(object value);
    void IList.RemoveAt(int index);
    bool IList.IsFixedSize { get; }
    bool IList.IsReadOnly { get; }
    bool ICollection.IsSynchronized { get; }
    public int Length { get; }
    public long LongLength {get;}
    public int Rank { get; }
    object ICollection.SyncRoot { get; }
    int ICollection.Count { get; }
    public virtual object this[int index] { get; set; }
}
```

```
// Namespace: System.Collections, Library: BCL
public class ArrayList: ICloneable, ICollection, IEnumerable, IList
{
   public ArrayList();
   public ArrayList(int capacity);
   public ArrayList(ICollection c);
   public static ArrayList Adapter(IList list);
   public virtual int Add(object value);
   public virtual void AddRange(ICollection c);
   public virtual int BinarySearch(object value, IComparer comparer);
   public virtual int BinarySearch(object value);
   public virtual int BinarySearch(int index, int count, object value,
      IComparer comparer);
   public virtual void Clear();
   public virtual object Clone();
   public virtual bool Contains(object item);
   public virtual void CopyTo(Array array, int arrayIndex);
   public virtual void CopyTo(int index, Array array, int arrayIndex, int
      count);
   public virtual void CopyTo(Array array);
   public static ArrayList FixedSize(ArrayList list);
   public virtual IEnumerator GetEnumerator();
   public virtual IEnumerator GetEnumerator(int index, int count);
   public virtual ArrayList GetRange(int index, int count);
   public virtual int IndexOf(object value);
   public virtual int IndexOf(object value, int startIndex, int count);
   public virtual int IndexOf(object value, int startIndex);
   public virtual void Insert(int index, object value);
   public virtual void InsertRange(int index, ICollection c);
   public virtual int LastIndexOf(object value, int startIndex, int count);
   public virtual int LastIndexOf(object value, int startIndex);
   public virtual int LastIndexOf(object value);
   public static ArrayList ReadOnly(ArrayList list);
   public virtual void Remove(object obj);
   public virtual void RemoveAt(int index);
   public virtual void RemoveRange(int index, int count);
   public static ArrayList Repeat(object value, int count);
   public virtual void Reverse(int index, int count);
   public virtual void Reverse();
   public virtual void SetRange(int index, ICollection c);
   public virtual void Sort(int index, int count, IComparer comparer);
   public virtual void Sort(IComparer comparer);
   public virtual void Sort();
   public static ArrayList Synchronized(ArrayList list);
```

```
    public virtual Array ToArray(Type type);
    public virtual object[] ToArray();
    public virtual void TrimToSize();
    public virtual int Capacity { get; set; }
    int ICollection.Count { get; }
    public virtual int Count { get; }
    bool IList.IsFixedSize { get; }
    public virtual bool IsFixedSize { get; }
    bool IList.IsReadOnly { get; }
    public virtual bool IsReadOnly { get; }
    bool ICollection.IsSynchronized { get; }
    public virtual bool IsSynchronized { get; }
    public virtual object this[int index] { get; set; }
    object ICollection.SyncRoot { get; }
    public virtual object SyncRoot { get; }
}

// Namespace: System, Library: BCL
public class ArrayTypeMismatchException: SystemException
{
    public ArrayTypeMismatchException();
    public ArrayTypeMismatchException(string message);
    public ArrayTypeMismatchException(string message, Exception innerException);
}

// Namespace: System.Text, Library: BCL
public class ASCIIEncoding: Encoding
{
    public ASCIIEncoding();
    public override int GetByteCount(char[] chars, int index, int count);
    public override int GetByteCount(string chars);
    public override int GetBytes(string chars, int charIndex, int charCount,
        byte[] bytes, int byteIndex);
    public override int GetBytes(char[] chars, int charIndex, int charCount,
        byte[] bytes, int byteIndex);
    public override int GetCharCount(byte[] bytes, int index, int count);
    public override int GetChars(byte[] bytes, int byteIndex, int byteCount,
        char[] chars, int charIndex);
    public override int GetMaxByteCount(int charCount);
    public override int GetMaxCharCount(int byteCount);
    public override string GetString(byte[] bytes, int byteIndex, int
        byteCount);
    public override string GetString(byte[] bytes);
}
```

```
// Namespace: System, Library: BCL
public delegate void AsyncCallback(IAsyncResult ar);

// Namespace: System, Library: BCL
public abstract class Attribute
{
   protected Attribute();
   public override bool Equals(object obj);
   public override int GetHashCode();
}

// Namespace: System, Library: BCL
public enum AttributeTargets
{
   All = Assembly | 0x2 | Class | Struct | Enum | Constructor | Method |
      Property | Field | Event | Interface | Parameter | Delegate |
      ReturnValue,
   Assembly = 0x1,
   Class = 0x4,
   Constructor = 0x20,
   Delegate = 0x1000,
   Enum = 0x10,
   Event = 0x200,
   Field = 0x100,
   Interface = 0x400,
   Method = 0x40,
   Parameter = 0x800,
   Property = 0x80,
   ReturnValue = 0x2000,
   Struct = 0x8,
}

// Namespace: System, Library: BCL
public sealed class AttributeUsageAttribute: Attribute
{
   public AttributeUsageAttribute(AttributeTargets validOn);
   public bool AllowMultiple { get; set; }
   public bool Inherited { get; set; }
   public AttributeTargets ValidOn { get; }
}

// Namespace: System, Library: BCL
public struct Boolean: IComparable
```

```
{
   public static readonly string FalseString;
   public static readonly string TrueString;
   public int CompareTo(object obj);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static bool Parse(string value);
   public string ToString(IFormatProvider provider);
   public override string ToString();
}

// Namespace: System, Library: BCL
public struct Byte: IComparable, IFormattable
{
   public const byte MaxValue = 255;
   public const byte MinValue = 0;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static byte Parse(string s);
   public static byte Parse(string s, NumberStyles style);
   public static byte Parse(string s, IFormatProvider provider);
   public static byte Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public struct Char: IComparable
{
   public const char MaxValue = (char)0xFFFF;
   public const char MinValue = (char)0x0;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static double GetNumericValue(char c);
   public static double GetNumericValue(string s, int index);
   public static UnicodeCategory GetUnicodeCategory(char c);
   public static UnicodeCategory GetUnicodeCategory(string s, int index);
   public static bool IsControl(char c);
   public static bool IsControl(string s, int index);
```

```
   public static bool IsDigit(char c);
   public static bool IsDigit(string s, int index);
   public static bool IsLetter(char c);
   public static bool IsLetter(string s, int index);
   public static bool IsLetterOrDigit(char c);
   public static bool IsLetterOrDigit(string s, int index);
   public static bool IsLower(char c);
   public static bool IsLower(string s, int index);
   public static bool IsNumber(char c);
   public static bool IsNumber(string s, int index);
   public static bool IsPunctuation(char c);
   public static bool IsPunctuation(string s, int index);
   public static bool IsSeparator(char c);
   public static bool IsSeparator(string s, int index);
   public static bool IsSurrogate(char c);
   public static bool IsSurrogate(string s, int index);
   public static bool IsSymbol(char c);
   public static bool IsSymbol(string s, int index);
   public static bool IsUpper(char c);
   public static bool IsUpper(string s, int index);
   public static bool IsWhiteSpace(char c);
   public static bool IsWhiteSpace(string s, int index);
   public static char Parse(string s);
   public static char ToLower(char c);
   public string ToString(IFormatProvider provider);
   public override string ToString();
   public static char ToUpper(char c);
}

// Namespace: System, Library: BCL
public sealed class CharEnumerator: ICloneable, IEnumerator
{
   public object Clone();
   public bool MoveNext();
   public void Reset();
   public char Current { get; }
   object IEnumerator.Current { get; }
}

// Namespace: System, Library: BCL
public sealed class CLSCompliantAttribute: Attribute
{
   public CLSCompliantAttribute(bool isCompliant);
   public bool IsCompliant { get; }
```

```
}

// Namespace: System.Security, Library: BCL
public abstract class CodeAccessPermission: IPermission
{
   protected CodeAccessPermission();
   public void Assert();
   public abstract IPermission Copy();
   public void Demand();
   public void Deny();
   public abstract void FromXml(SecurityElement elem);
   public abstract IPermission Intersect(IPermission target);
   public abstract bool IsSubsetOf(IPermission target);
   public override string ToString();
   public abstract SecurityElement ToXml();
   public virtual IPermission Union(IPermission other);
}

// Namespace: System.Security.Permissions, Library: BCL
public abstract class CodeAccessSecurityAttribute: SecurityAttribute
{
   protected CodeAccessSecurityAttribute();
   public CodeAccessSecurityAttribute(SecurityAction action);
}

// Namespace: System.Collections, Library: BCL
public sealed class Comparer: IComparer
{
   public static readonly Comparer Default;
   public int Compare(object a, object b);
}

// Namespace: System.Diagnostics, Library: BCL
public sealed class ConditionalAttribute: Attribute
{
   public ConditionalAttribute(string conditionString);
   public string ConditionString { get; }
}

// Namespace: System, Library: BCL
public sealed class Console
{
   public static Stream OpenStandardError();
   public static Stream OpenStandardError(int bufferSize);
```

```
    public static Stream OpenStandardInput();
    public static Stream OpenStandardInput(int bufferSize);
    public static Stream OpenStandardOutput();
    public static Stream OpenStandardOutput(int bufferSize);
    public static int Read();
    public static string ReadLine();
    public static void SetError(TextWriter newError);
    public static void SetIn(TextReader newIn);
    public static void SetOut(TextWriter newOut);
    public static void Write(string format, object arg0);
    public static void Write(string format, object arg0, object arg1);
    public static void Write(string format, object arg0, object arg1, object
       arg2);
    public static void Write(string format, params object[] arg);
    public static void Write(bool value);
    public static void Write(char value);
    public static void Write(char[] buffer);
    public static void Write(char[] buffer, int index, int count);
    public static void Write(double value);
    public static void Write(decimal value);
    public static void Write(float value);
    public static void Write(int value);
    public static void Write(uint value);
    public static void Write(long value);
    public static void Write(ulong value);
    public static void Write(object value);
    public static void Write(string value);
    public static void WriteLine();
    public static void WriteLine(bool value);
    public static void WriteLine(char value);
    public static void WriteLine(char[] buffer);
    public static void WriteLine(char[] buffer, int index, int count);
    public static void WriteLine(decimal value);
    public static void WriteLine(double value);
    public static void WriteLine(float value);
    public static void WriteLine(int value);
    public static void WriteLine(uint value);
    public static void WriteLine(long value);
    public static void WriteLine(ulong value);
    public static void WriteLine(object value);
    public static void WriteLine(string value);
    public static void WriteLine(string format, object arg0);
    public static void WriteLine(string format, object arg0, object arg1);
    public static void WriteLine(string format, object arg0, object arg1,
```

```
        object arg2);
    public static void WriteLine(string format, params object[] arg);
    public static TextWriter Error { get; }
    public static TextReader In { get; }
    public static TextWriter Out { get; }
}

// Namespace: System, Library: BCL
public sealed class Convert
{
    public static bool ToBoolean(bool value);
    public static bool ToBoolean(sbyte value);
    public static bool ToBoolean(byte value);
    public static bool ToBoolean(short value);
    public static bool ToBoolean(ushort value);
    public static bool ToBoolean(int value);
    public static bool ToBoolean(uint value);
    public static bool ToBoolean(long value);
    public static bool ToBoolean(ulong value);
    public static bool ToBoolean(string value);
    public static bool ToBoolean(float value);
    public static bool ToBoolean(double value);
    public static bool ToBoolean(decimal value);
    public static byte ToByte(bool value);
    public static byte ToByte(byte value);
    public static byte ToByte(char value);
    public static byte ToByte(sbyte value);
    public static byte ToByte(short value);
    public static byte ToByte(ushort value);
    public static byte ToByte(int value);
    public static byte ToByte(uint value);
    public static byte ToByte(long value);
    public static byte ToByte(ulong value);
    public static byte ToByte(float value);
    public static byte ToByte(double value);
    public static byte ToByte(decimal value);
    public static byte ToByte(string value);
    public static byte ToByte(string value, IFormatProvider provider);
    public static char ToChar(char value);
    public static char ToChar(sbyte value);
    public static char ToChar(byte value);
    public static char ToChar(short value);
    public static char ToChar(ushort value);
    public static char ToChar(int value);
```

```
public static char ToChar(uint value);
public static char ToChar(long value);
public static char ToChar(ulong value);
public static char ToChar(string value);
public static DateTime ToDateTime(DateTime value);
public static DateTime ToDateTime(string value);
public static DateTime ToDateTime(string value, IFormatProvider provider);
public static decimal ToDecimal(sbyte value);
public static decimal ToDecimal(byte value);
public static decimal ToDecimal(short value);
public static decimal ToDecimal(ushort value);
public static decimal ToDecimal(int value);
public static decimal ToDecimal(uint value);
public static decimal ToDecimal(long value);
public static decimal ToDecimal(ulong value);
public static decimal ToDecimal(float value);
public static decimal ToDecimal(double value);
public static decimal ToDecimal(string value);
public static decimal ToDecimal(string value, IFormatProvider provider);
public static decimal ToDecimal(decimal value);
public static decimal ToDecimal(bool value);
public static double ToDouble(sbyte value);
public static double ToDouble(byte value);
public static double ToDouble(short value);
public static double ToDouble(ushort value);
public static double ToDouble(int value);
public static double ToDouble(uint value);
public static double ToDouble(long value);
public static double ToDouble(ulong value);
public static double ToDouble(float value);
public static double ToDouble(double value);
public static double ToDouble(decimal value);
public static double ToDouble(string value);
public static double ToDouble(string value, IFormatProvider provider);
public static double ToDouble(bool value);
public static short ToInt16(bool value);
public static short ToInt16(char value);
public static short ToInt16(sbyte value);
public static short ToInt16(byte value);
public static short ToInt16(ushort value);
public static short ToInt16(int value);
public static short ToInt16(uint value);
public static short ToInt16(short value);
public static short ToInt16(long value);
```

```
        public static short ToInt16(ulong value);
        public static short ToInt16(float value);
        public static short ToInt16(double value);
        public static short ToInt16(decimal value);
        public static short ToInt16(string value);
        public static short ToInt16(string value, IFormatProvider provider);
        public static int ToInt32(bool value);
        public static int ToInt32(char value);
        public static int ToInt32(sbyte value);
        public static int ToInt32(byte value);
        public static int ToInt32(short value);
        public static int ToInt32(ushort value);
        public static int ToInt32(uint value);
        public static int ToInt32(int value);
        public static int ToInt32(long value);
        public static int ToInt32(ulong value);
        public static int ToInt32(float value);
        public static int ToInt32(double value);
        public static int ToInt32(decimal value);
        public static int ToInt32(string value);
        public static int ToInt32(string value, IFormatProvider provider);
        public static long ToInt64(bool value);
        public static long ToInt64(char value);
        public static long ToInt64(sbyte value);
        public static long ToInt64(byte value);
        public static long ToInt64(short value);
        public static long ToInt64(ushort value);
        public static long ToInt64(int value);
        public static long ToInt64(uint value);
        public static long ToInt64(ulong value);
        public static long ToInt64(long value);
        public static long ToInt64(float value);
        public static long ToInt64(double value);
        public static long ToInt64(decimal value);
        public static long ToInt64(string value);
        public static long ToInt64(string value, IFormatProvider provider);
        public static sbyte ToSByte(bool value);
        public static sbyte ToSByte(sbyte value);
        public static sbyte ToSByte(char value);
        public static sbyte ToSByte(byte value);
        public static sbyte ToSByte(short value);
        public static sbyte ToSByte(ushort value);
        public static sbyte ToSByte(int value);
        public static sbyte ToSByte(uint value);
```

```
public static sbyte ToSByte(long value);
public static sbyte ToSByte(ulong value);
public static sbyte ToSByte(float value);
public static sbyte ToSByte(double value);
public static sbyte ToSByte(decimal value);
public static sbyte ToSByte(string value);
public static sbyte ToSByte(string value, IFormatProvider provider);
public static float ToSingle(sbyte value);
public static float ToSingle(byte value);
public static float ToSingle(short value);
public static float ToSingle(ushort value);
public static float ToSingle(int value);
public static float ToSingle(uint value);
public static float ToSingle(long value);
public static float ToSingle(ulong value);
public static float ToSingle(float value);
public static float ToSingle(double value);
public static float ToSingle(decimal value);
public static float ToSingle(string value);
public static float ToSingle(string value, IFormatProvider provider);
public static float ToSingle(bool value);
public static string ToString(bool value);
public static string ToString(char value);
public static string ToString(sbyte value);
public static string ToString(sbyte value, IFormatProvider provider);
public static string ToString(byte value);
public static string ToString(byte value, IFormatProvider provider);
public static string ToString(short value);
public static string ToString(short value, IFormatProvider provider);
public static string ToString(ushort value);
public static string ToString(ushort value, IFormatProvider provider);
public static string ToString(int value);
public static string ToString(int value, IFormatProvider provider);
public static string ToString(uint value);
public static string ToString(uint value, IFormatProvider provider);
public static string ToString(long value);
public static string ToString(long value, IFormatProvider provider);
public static string ToString(ulong value);
public static string ToString(ulong value, IFormatProvider provider);
public static string ToString(float value);
public static string ToString(float value, IFormatProvider provider);
public static string ToString(double value);
public static string ToString(double value, IFormatProvider provider);
public static string ToString(decimal value);
```

```
public static string ToString(decimal value, IFormatProvider provider);
public static string ToString(DateTime value);
public static string ToString(DateTime value, IFormatProvider provider);
public static string ToString(string value);
public static ushort ToUInt16(bool value);
public static ushort ToUInt16(char value);
public static ushort ToUInt16(sbyte value);
public static ushort ToUInt16(byte value);
public static ushort ToUInt16(short value);
public static ushort ToUInt16(int value);
public static ushort ToUInt16(ushort value);
public static ushort ToUInt16(uint value);
public static ushort ToUInt16(long value);
public static ushort ToUInt16(ulong value);
public static ushort ToUInt16(float value);
public static ushort ToUInt16(double value);
public static ushort ToUInt16(decimal value);
public static ushort ToUInt16(string value);
public static ushort ToUInt16(string value, IFormatProvider provider);
public static uint ToUInt32(bool value);
public static uint ToUInt32(char value);
public static uint ToUInt32(sbyte value);
public static uint ToUInt32(byte value);
public static uint ToUInt32(short value);
public static uint ToUInt32(ushort value);
public static uint ToUInt32(int value);
public static uint ToUInt32(uint value);
public static uint ToUInt32(long value);
public static uint ToUInt32(ulong value);
public static uint ToUInt32(float value);
public static uint ToUInt32(double value);
public static uint ToUInt32(decimal value);
public static uint ToUInt32(string value);
public static uint ToUInt32(string value, IFormatProvider provider);
public static ulong ToUInt64(bool value);
public static ulong ToUInt64(char value);
public static ulong ToUInt64(sbyte value);
public static ulong ToUInt64(byte value);
public static ulong ToUInt64(short value);
public static ulong ToUInt64(ushort value);
public static ulong ToUInt64(int value);
public static ulong ToUInt64(uint value);
public static ulong ToUInt64(long value);
public static ulong ToUInt64(ulong value);
```

```
    public static ulong ToUInt64(float value);
    public static ulong ToUInt64(double value);
    public static ulong ToUInt64(decimal value);
    public static ulong ToUInt64(string value);
    public static ulong ToUInt64(string value, IFormatProvider provider);
}

// Namespace: System, Library: BCL
public struct DateTime: IComparable, IFormattable
{
    public DateTime(long ticks);
    public DateTime(int year, int month, int day);
    public DateTime(int year, int month, int day, int hour, int minute, int
        second);
    public DateTime(int year, int month, int day, int hour, int minute, int
        second, int millisecond);
    public static readonly DateTime MaxValue;
    public static readonly DateTime MinValue;
    public DateTime Add(TimeSpan value);
    public DateTime AddDays(double value);
    public DateTime AddHours(double value);
    public DateTime AddMilliseconds(double value);
    public DateTime AddMinutes(double value);
    public DateTime AddMonths(int months);
    public DateTime AddSeconds(double value);
    public DateTime AddTicks(long value);
    public DateTime AddYears(int value);
    public static int Compare(DateTime t1, DateTime t2);
    public int CompareTo(object value);
    public static int DaysInMonth(int year, int month);
    public override bool Equals(object value);
    public static bool Equals(DateTime t1, DateTime t2);
    public override int GetHashCode();
    public static bool IsLeapYear(int year);
    public static DateTime operator +(DateTime d, TimeSpan t);
    public static bool operator ==(DateTime d1, DateTime d2);
    public static bool operator >(DateTime t1, DateTime t2);
    public static bool operator >=(DateTime t1, DateTime t2);
    public static bool operator !=(DateTime d1, DateTime d2);
    public static bool operator  <(DateTime t1, DateTime t2);
    public static bool operator <=(DateTime t1, DateTime t2);
    public static DateTime operator -(DateTime d, TimeSpan t);
    public static TimeSpan operator -(DateTime d1, DateTime d2);
    public static DateTime Parse(string s);
```

```
    public static DateTime Parse(string s, IFormatProvider provider);
    public static DateTime Parse(string s, IFormatProvider provider,
       DateTimeStyles styles);
    public static DateTime ParseExact(string s, string format, IFormatProvider
       provider);
    public static DateTime ParseExact(string s, string format, IFormatProvider
       provider, DateTimeStyles style);
    public static DateTime ParseExact(string s, string[] formats,
       IFormatProvider provider, DateTimeStyles style);
    public TimeSpan Subtract(DateTime value);
    public DateTime Subtract(TimeSpan value);
    public DateTime ToLocalTime();
    public string ToLongDateString();
    public string ToLongTimeString();
    public string ToShortDateString();
    public string ToShortTimeString();
    public string ToString(IFormatProvider provider);
    public string ToString(string format, IFormatProvider provider);
    public override string ToString();
    public string ToString(string format);
    public DateTime ToUniversalTime();
    public DateTime Date { get; }
    public int Day { get; }
    public int DayOfYear { get; }
    public int Hour { get; }
    public int Millisecond { get; }
    public int Minute { get; }
    public int Month { get; }
    public static DateTime Now { get; }
    public int Second { get; }
    public long Ticks { get; }
    public TimeSpan TimeOfDay { get; }
    public static DateTime Today { get; }
    public static DateTime UtcNow { get; }
    public int Year { get; }
}

// Namespace: System.Globalization, Library: BCL
public sealed class DateTimeFormatInfo: ICloneable, IFormatProvider
{
    public DateTimeFormatInfo();
    public object Clone();
    public string GetAbbreviatedMonthName(int month);
    public int GetEra(string eraName);
```

```
    public string GetEraName(int era);
    public object GetFormat(Type formatType);
    public string GetMonthName(int month);
    public static DateTimeFormatInfo ReadOnly(DateTimeFormatInfo dtfi);
    public string[] AbbreviatedDayNames { get; set; }
    public string[] AbbreviatedMonthNames { get; set; }
    public string AMDesignator { get; set; }
    public static DateTimeFormatInfo CurrentInfo { get; }
    public string DateSeparator { get; set; }
    public string[] DayNames { get; set; }
    public string FullDateTimePattern { get; set; }
    public static DateTimeFormatInfo InvariantInfo { get; }
    public bool IsReadOnly { get; }
    public string LongDatePattern { get; set; }
    public string LongTimePattern { get; set; }
    public string MonthDayPattern { get; set; }
    public string[] MonthNames { get; set; }
    public string PMDesignator { get; set; }
    public string ShortDatePattern { get; set; }
    public string ShortTimePattern { get; set; }
    public string TimeSeparator { get; set; }
    public string YearMonthPattern { get; set; }
}

// Namespace: System.Globalization, Library: BCL
public enum DateTimeStyles
{
    AdjustToUniversal = 0x10,
    AllowInnerWhite = 0x4,
    AllowLeadingWhite = 0x1,
    AllowTrailingWhite = 0x2,
    AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite,
    NoCurrentDateDefault = 0x8,
    None = 0x0,
}

// Namespace: System, Library: ExtendedNumerics
public struct Decimal: IComparable, IFormattable
{
    public Decimal(int value);
    public Decimal(uint value);
    public Decimal(long value);
    public Decimal(ulong value);
    public Decimal(float value);
```

```
public Decimal(double value);
public Decimal(int[] bits);
public static readonly decimal MaxValue;
public static readonly decimal MinusOne;
public static readonly decimal MinValue;
public static readonly decimal One;
public static readonly decimal Zero;
public static decimal Add(decimal d1, decimal d2);
public static int Compare(decimal d1, decimal d2);
public int CompareTo(object value);
public static decimal Divide(decimal d1, decimal d2);
public override bool Equals(object value);
public static bool Equals(decimal d1, decimal d2);
public static decimal Floor(decimal d);
public static int[] GetBits(decimal d);
public override int GetHashCode();
public static decimal Multiply(decimal d1, decimal d2);
public static decimal Negate(decimal d);
public static Decimal operator +(Decimal d1, Decimal d2);
public static Decimal operator --(Decimal d);
public static Decimal operator /(Decimal d1, Decimal d2);
public static bool operator ==(Decimal d1, Decimal d2);
public static explicit operator Decimal(float value);
public static explicit operator Decimal(double value);
public static explicit operator byte(Decimal value);
public static explicit operator sbyte(Decimal value);
public static explicit operator char(Decimal value);
public static explicit operator short(Decimal value);
public static explicit operator ushort(Decimal value);
public static explicit operator int(Decimal value);
public static explicit operator uint(Decimal value);
public static explicit operator long(Decimal value);
public static explicit operator ulong(Decimal value);
public static explicit operator float(Decimal value);
public static explicit operator double(Decimal value);
public static bool operator >(Decimal d1, Decimal d2);
public static bool operator >=(Decimal d1, Decimal d2);
public static implicit operator Decimal(byte value);
public static implicit operator Decimal(sbyte value);
public static implicit operator Decimal(short value);
public static implicit operator Decimal(ushort value);
public static implicit operator Decimal(char value);
public static implicit operator Decimal(int value);
public static implicit operator Decimal(uint value);
```

```
   public static implicit operator Decimal(long value);
   public static implicit operator Decimal(ulong value);
   public static Decimal operator ++(Decimal d);
   public static bool operator !=(Decimal d1, Decimal d2);
   public static bool operator  <(Decimal d1, Decimal d2);
   public static bool operator <=(Decimal d1, Decimal d2);
   public static Decimal operator %(Decimal d1, Decimal d2);
   public static Decimal operator *(Decimal d1, Decimal d2);
   public static Decimal operator -(Decimal d1, Decimal d2);
   public static Decimal operator -(Decimal d);
   public static Decimal operator +(Decimal d);
   public static decimal Parse(string s);
   public static decimal Parse(string s, NumberStyles style);
   public static decimal Parse(string s, IFormatProvider provider);
   public static decimal Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public static decimal Remainder(decimal d1, decimal d2);
   public static decimal Round(decimal d, int decimals);
   public static decimal Subtract(decimal d1, decimal d2);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
   public static decimal Truncate(decimal d);
}

// Namespace: System.Text, Library: BCL
public abstract class Decoder
{
   protected Decoder();
   public abstract int GetCharCount(byte[] bytes, int index, int count);
   public abstract int GetChars(byte[] bytes, int byteIndex, int byteCount,
      char[] chars, int charIndex);
}

// Namespace: System, Library: BCL
public abstract class Delegate: ICloneable
{
   public virtual object Clone();
   public static Delegate Combine(Delegate a, Delegate b);
   public static Delegate Combine(Delegate[] delegates);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public virtual Delegate[] GetInvocationList();
```

```
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);
    public object Target { get; }
}

// Namespace: System.Collections, Library: BCL
public struct DictionaryEntry
{
    public DictionaryEntry(object key, object value);
    public object Key { get; set; }
    public object Value { get; set; }
}

// Namespace: System.IO, Library: BCL
public sealed class Directory
{
    public static void Delete(string path);
    public static void Delete(string path, bool recursive);
    public static bool Exists(string path);
    public static DateTime GetCreationTime(string path);
    public static string GetCurrentDirectory();
    public static string[] GetDirectories(string path);
    public static string[] GetDirectories(string path, string searchPattern);
    public static string GetDirectoryRoot(string path);
    public static string[] GetFiles(string path);
    public static string[] GetFiles(string path, string searchPattern);
    public static string[] GetFileSystemEntries(string path);
    public static string[] GetFileSystemEntries(string path, string
        searchPattern);
    public static DateTime GetLastAccessTime(string path);
    public static DateTime GetLastWriteTime(string path);
    public static void Move(string sourceDirName, string destDirName);
    public static void SetCreationTime(string path, DateTime creationTime);
    public static void SetCurrentDirectory(string path);
    public static void SetLastAccessTime(string path, DateTime lastAccessTime);
    public static void SetLastWriteTime(string path, DateTime lastWriteTime);
}

// Namespace: System.IO, Library: BCL
public class DirectoryNotFoundException: IOException
{
    public DirectoryNotFoundException();
```

```
   public DirectoryNotFoundException(string message);
   public DirectoryNotFoundException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public class DivideByZeroException: ArithmeticException
{
   public DivideByZeroException();
   public DivideByZeroException(string message);
   public DivideByZeroException(string message, Exception innerException);
}

// Namespace: System, Library: ExtendedNumerics
public struct Double: IComparable, IFormattable
{
   public const double Epsilon = 4.9406564584124654e-324;
   public const double MaxValue = 1.7976931348623157e+308;
   public const double MinValue = -1.7976931348623157e+308;
   public const double NaN = (double)0.0 / (double)0.0;
   public const double NegativeInfinity = (double)-1.0 / (double)(0.0);
   public const double PositiveInfinity = (double)1.0 / (double)(0.0);
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static bool IsInfinity(double d);
   public static bool IsNaN(double d);
   public static bool IsNegativeInfinity(double d);
   public static bool IsPositiveInfinity(double d);
   public static double Parse(string s);
   public static double Parse(string s, NumberStyles style);
   public static double Parse(string s, IFormatProvider provider);
   public static double Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public class DuplicateWaitObjectException: ArgumentException
{
   public DuplicateWaitObjectException();
   public DuplicateWaitObjectException(string parameterName);
```

```
    public DuplicateWaitObjectException(string parameterName, string message);
}

// Namespace: System.Text, Library: BCL
public abstract class Encoder
{
    protected Encoder();
    public abstract int GetByteCount(char[] chars, int index, int count, bool
        flush);
    public abstract int GetBytes(char[] chars, int charIndex, int charCount,
        byte[] bytes, int byteIndex, bool flush);
}

// Namespace: System.Text, Library: BCL
public abstract class Encoding
{
    protected Encoding();
    public static byte[] Convert(Encoding srcEncoding, Encoding dstEncoding,
        byte[] bytes);
    public static byte[] Convert(Encoding srcEncoding, Encoding dstEncoding,
        byte[] bytes, int index, int count);
    public override bool Equals(object value);
    public abstract int GetByteCount(char[] chars, int index, int count);
    public virtual int GetByteCount(string s);
    public virtual int GetByteCount(char[] chars);
    public virtual int GetBytes(string s, int charIndex, int charCount, byte[]
        bytes, int byteIndex);
    public virtual byte[] GetBytes(string s);
    public abstract int GetBytes(char[] chars, int charIndex, int charCount,
        byte[] bytes, int byteIndex);
    public virtual byte[] GetBytes(char[] chars, int index, int count);
    public virtual byte[] GetBytes(char[] chars);
    public abstract int GetCharCount(byte[] bytes, int index, int count);
    public virtual int GetCharCount(byte[] bytes);
    public abstract int GetChars(byte[] bytes, int byteIndex, int byteCount,
        char[] chars, int charIndex);
    public virtual char[] GetChars(byte[] bytes, int index, int count);
    public virtual char[] GetChars(byte[] bytes);
    public virtual Decoder GetDecoder();
    public virtual Encoder GetEncoder();
    public override int GetHashCode();
    public abstract int GetMaxByteCount(int charCount);
    public abstract int GetMaxCharCount(int byteCount);
    public virtual byte[] GetPreamble();
```

```
   public virtual string GetString(byte[] bytes, int index, int count);
   public virtual string GetString(byte[] bytes);
   public static Encoding ASCII { get; }
   public static Encoding BigEndianUnicode { get; }
   public static Encoding Default { get; }
   public static Encoding Unicode { get; }
   public static Encoding UTF8 { get; }
}

// Namespace: System.IO, Library: BCL
public class EndOfStreamException: IOException
{
   public EndOfStreamException();
   public EndOfStreamException(string message);
   public EndOfStreamException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public abstract class Enum: ValueType, IComparable, IFormattable
{
   public int CompareTo(object target);
   public override bool Equals(object obj);
   public static string Format(Type enumType, object value, string format);
   public override int GetHashCode();
   public static string GetName(Type enumType, object value);
   public static string[] GetNames(Type enumType);
   public static Type GetUnderlyingType(Type enumType);
   public static Array GetValues(Type enumType);
   public static bool IsDefined(Type enumType, object value);
   public static object Parse(Type enumType, string value);
   public static object Parse(Type enumType, string value, bool ignoreCase);
   public static object ToObject(Type enumType, object value);
   public static object ToObject(Type enumType, sbyte value);
   public static object ToObject(Type enumType, short value);
   public static object ToObject(Type enumType, int value);
   public static object ToObject(Type enumType, byte value);
   public static object ToObject(Type enumType, ushort value);
   public static object ToObject(Type enumType, uint value);
   public static object ToObject(Type enumType, long value);
   public static object ToObject(Type enumType, ulong value);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
```

```
}

// Namespace: System, Library: BCL
public sealed class Environment
{
   public static void Exit(int exitCode);
   public static string[] GetCommandLineArgs();
   public static string GetEnvironmentVariable(string variable);
   public static IDictionary GetEnvironmentVariables();
   public static string CommandLine { get; }
   public static int ExitCode { get; set; }
   public bool HasShutdownStarted { get; }
   public static string NewLine { get; }
   public static string StackTrace { get; }
   public static int TickCount { get; }
   public static Version Version { get; }
}

// Namespace: System.Security.Permissions, Library: BCL
public sealed class EnvironmentPermission: CodeAccessPermission
{
   public EnvironmentPermission(PermissionState state);
   public EnvironmentPermission(EnvironmentPermissionAccess flag, string
      pathList);
   public override IPermission Copy();
   public override void FromXml(SecurityElement esd);
   public override IPermission Intersect(IPermission target);
   public override bool IsSubsetOf(IPermission target);
   public override SecurityElement ToXml();
   public override IPermission Union(IPermission other);
}

// Namespace: System.Security.Permissions, Library: BCL
public enum EnvironmentPermissionAccess
{
   AllAccess = Read | Write,
   NoAccess = 0x0,
   Read = 0x1,
   Write = 0x2,
}

// Namespace: System.Security.Permissions, Library: BCL
public sealed class EnvironmentPermissionAttribute: CodeAccessSecurityAttribute
{
```

```
   public EnvironmentPermissionAttribute(SecurityAction action);
   public override IPermission CreatePermission();
   public string All { set; }
   public string Read { get; set; }
   public string Write { get; set; }
}

// Namespace: System, Library: BCL
public class EventArgs
{
   public EventArgs();
   public static readonly EventArgs Empty;
}

// Namespace: System, Library: BCL
public delegate void EventHandler(object sender, EventArgs e);

// Namespace: System, Library: BCL
public class Exception
{
   public Exception();
   public Exception(string message);
   public Exception(string message, Exception innerException);
   public virtual Exception GetBaseException();
   public override string ToString();
   public Exception InnerException { get; }
   public virtual string Message { get; }
   public virtual string StackTrace { get; }
}

// Namespace: System, Library: BCL
public sealed class ExecutionEngineException: SystemException
{
   public ExecutionEngineException();
   public ExecutionEngineException(string message);
   public ExecutionEngineException(string message, Exception innerException);
}

// Namespace: System.IO, Library: BCL
public sealed class File
{
   public static StreamWriter AppendText(string path);
   public static void Copy(string sourceFileName, string destFileName);
   public static void Copy(string sourceFileName, string destFileName, bool
```

```
        overwrite);
    public static FileStream Create(string path);
    public static FileStream Create(string path, int bufferSize);
    public static StreamWriter CreateText(string path);
    public static void Delete(string path);
    public static bool Exists(string path);
    public static DateTime GetCreationTime(string path);
    public static DateTime GetLastAccessTime(string path);
    public static DateTime GetLastWriteTime(string path);
    public static void Move(string sourceFileName, string destFileName);
    public static FileStream Open(string path, FileMode mode);
    public static FileStream Open(string path, FileMode mode, FileAccess
        access);
    public static FileStream Open(string path, FileMode mode, FileAccess
        access, FileShare share);
    public static FileStream OpenRead(string path);
    public static StreamReader OpenText(string path);
    public static FileStream OpenWrite(string path);
    public static void SetCreationTime(string path, DateTime creationTime);
    public static void SetLastAccessTime(string path, DateTime lastAccessTime);
    public static void SetLastWriteTime(string path, DateTime lastWriteTime);
}

// Namespace: System.IO, Library: BCL
public enum FileAccess
{
    Read = 0x1,
    ReadWrite = Read | Write,
    Write = 0x2,
}

// Namespace: System.Security.Permissions, Library: BCL
public sealed class FileIOPermission: CodeAccessPermission
{
    public FileIOPermission(PermissionState state);
    public FileIOPermission(FileIOPermissionAccess access, string path);
    public override IPermission Copy();
    public override void FromXml(SecurityElement esd);
    public override IPermission Intersect(IPermission target);
    public override bool IsSubsetOf(IPermission target);
    public override SecurityElement ToXml();
    public override IPermission Union(IPermission other);
}
```

```
// Namespace: System.Security.Permissions, Library: BCL
public enum FileIOPermissionAccess
{
   AllAccess = Read | Write | Append | PathDiscovery,
   Append = 0x4,
   NoAccess = 0x0,
   PathDiscovery = 0x8,
   Read = 0x1,
   Write = 0x2,
}

// Namespace: System.Security.Permissions, Library: BCL
public sealed class FileIOPermissionAttribute: CodeAccessSecurityAttribute
{
   public FileIOPermissionAttribute(SecurityAction action);
   public override IPermission CreatePermission();
   public string All { set; }
   public string Append { get; set; }
   public string PathDiscovery { get; set; }
   public string Read { get; set; }
   public string Write { get; set; }
}

// Namespace: System.IO, Library: BCL
public class FileLoadException: IOException
{
   public FileLoadException();
   public FileLoadException(string message);
   public FileLoadException(string message, Exception inner);
   public FileLoadException(string message, string fileName);
   public FileLoadException(string message, string fileName, Exception inner);
   public override string ToString();
   public string FileName { get; }
   public override string Message { get; }
}

// Namespace: System.IO, Library: BCL
public enum FileMode
{
   Append = 6,
   Create = 2,
   CreateNew = 1,
   Open = 3,
   OpenOrCreate = 4,
```

```
    Truncate = 5,
}

// Namespace: System.IO, Library: BCL
public class FileNotFoundException: IOException
{
    public FileNotFoundException();
    public FileNotFoundException(string message);
    public FileNotFoundException(string message, Exception innerException);
    public FileNotFoundException(string message, string fileName);
    public FileNotFoundException(string message, string fileName, Exception
        innerException);
    public override string ToString();
    public string FileName { get; }
    public override string Message { get; }
}

// Namespace: System.IO, Library: BCL
public enum FileShare
{
    None = 0x0,
    Read = 0x1,
    ReadWrite = Read | Write,
    Write = 0x2,
}

// Namespace: System.IO, Library: BCL
public class FileStream: Stream
{
    public FileStream(string path, FileMode mode);
    public FileStream(string path, FileMode mode, FileAccess access);
    public FileStream(string path, FileMode mode, FileAccess access, FileShare
        share);
    public FileStream(string path, FileMode mode, FileAccess access, FileShare
        share, int bufferSize);
    public FileStream(string path, FileMode mode, FileAccess access, FileShare
        share, int bufferSize, bool useAsync);
    public override IAsyncResult BeginRead(byte[] array, int offset, int
        numBytes, AsyncCallback userCallback, object stateObject);
    public override IAsyncResult BeginWrite(byte[] array, int offset, int
        numBytes, AsyncCallback userCallback, object stateObject);
    public override void Close();
    protected virtual void Dispose(bool disposing);
    public override int EndRead(IAsyncResult asyncResult);
```

```
   public override void EndWrite(IAsyncResult asyncResult);
   ~FileStream();
   public override void Flush();
   public override int Read(byte[] array, int offset, int count);
   public override int ReadByte();
   public override long Seek(long offset, SeekOrigin origin);
   public override void SetLength(long value);
   public override void Write(byte[] array, int offset, int count);
   public override void WriteByte(byte value);
   public override bool CanRead { get; }
   public override bool CanSeek { get; }
   public override bool CanWrite { get; }
   public virtual bool IsAsync { get; }
   public override long Length { get; }
   public override long Position { get; set; }
}

// Namespace: System, Library: BCL
public class FlagsAttribute: Attribute
{
   public FlagsAttribute();
}

// Namespace: System, Library: BCL
public class FormatException: SystemException
{
   public FormatException();
   public FormatException(string message);
   public FormatException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public sealed class GC
{
   public static void KeepAlive(object obj);
   public static void ReRegisterForFinalize(object obj);
   public static void SuppressFinalize(object obj);
   public static void WaitForPendingFinalizers();
}

// Namespace: System.Collections, Library: BCL
public class Hashtable: ICloneable, ICollection, IDictionary, IEnumerable
{
   public Hashtable();
```

```
    public Hashtable(int capacity);
    public Hashtable(IHashCodeProvider hcp, IComparer comparer);
    public Hashtable(int capacity, IHashCodeProvider hcp, IComparer comparer);
    public Hashtable(IDictionary d);
    public Hashtable(IDictionary d, IHashCodeProvider hcp, IComparer comparer);
    public virtual void Add(object key, object value);
    public virtual void Clear();
    public virtual object Clone();
    public virtual bool Contains(object key);
    public virtual bool ContainsKey(object key);
    public virtual bool ContainsValue(object value);
    public virtual void CopyTo(Array array, int arrayIndex);
    public virtual IDictionaryEnumerator GetEnumerator();
    protected virtual int GetHash(object key);
    protected virtual bool KeyEquals(object item, object key);
    public virtual void Remove(object key);
    public static Hashtable Synchronized(Hashtable table);
    IEnumerator IEnumerable.GetEnumerator();
    int ICollection.Count { get; }
    public virtual int Count { get; }
    bool IDictionary.IsFixedSize { get; }
    public virtual bool IsFixedSize { get; }
    bool IDictionary.IsReadOnly { get; }
    public virtual bool IsReadOnly { get; }
    bool ICollection.IsSynchronized { get; }
    public virtual bool IsSynchronized { get; }
    public virtual object this[object key] { get; set; }
    ICollection IDictionary.Keys { get; }
    public virtual ICollection Keys { get; }
    object ICollection.SyncRoot { get; }
    public virtual object SyncRoot { get; }
    ICollection IDictionary.Values { get; }
    public virtual ICollection Values { get; }
}

// Namespace: System, Library: BCL
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```

```
// Namespace: System, Library: BCL
public interface ICloneable
{
   object Clone();
}

// Namespace: System.Collections, Library: BCL
public interface ICollection: IEnumerable
{
   void CopyTo(Array array, int index);
   int Count { get; }
   bool IsSynchronized { get; }
   object SyncRoot { get; }
}

// Namespace: System, Library: BCL
public interface IComparable
{
   int CompareTo(object obj);
}

// Namespace: System.Collections, Library: BCL
public interface IComparer
{
   int Compare(object x, object y);
}

// Namespace: System.Collections, Library: BCL
public interface IDictionary: ICollection, IEnumerable
{
   void Add(object key, object value);
   void Clear();
   bool Contains(object key);
   IDictionaryEnumerator GetEnumerator();
   void Remove(object key);
   bool IsFixedSize { get; }
   bool IsReadOnly { get; }
   object this[object key] { get; set; }
   ICollection Keys { get; }
   ICollection Values { get; }
}

// Namespace: System.Collections, Library: BCL
public interface IDictionaryEnumerator: IEnumerator
```

```
{
   DictionaryEntry Entry { get; }
   object Key { get; }
   object Value { get; }
}

// Namespace: System, Library: BCL
public interface IDisposable
{
   void Dispose();
}

// Namespace: System.Collections, Library: BCL
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}

// Namespace: System.Collections, Library: BCL
public interface IEnumerator
{
   bool MoveNext();
   void Reset();
   object Current { get; }
}

// Namespace: System, Library: BCL
public interface IFormatProvider
{
   object GetFormat(Type formatType);
}

// Namespace: System, Library: BCL
public interface IFormattable
{
   string ToString(string format, IFormatProvider formatProvider);
}

// Namespace: System.Collections, Library: BCL
public interface IHashCodeProvider
{
   int GetHashCode(object obj);
}
```

```
// Namespace: System.Collections, Library: BCL
public interface IList: ICollection, IEnumerable
{
   int Add(object value);
   void Clear();
   bool Contains(object value);
   int IndexOf(object value);
   void Insert(int index, object value);
   void Remove(object value);
   void RemoveAt(int index);
   bool IsFixedSize { get; }
   bool IsReadOnly { get; }
   object this[int index] { get; set; }
}

// Namespace: System, Library: BCL
public sealed class IndexOutOfRangeException: SystemException
{
   public IndexOutOfRangeException();
   public IndexOutOfRangeException(string message);
   public IndexOutOfRangeException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public struct Int16: IComparable, IFormattable
{
   public const short MaxValue = 32767;
   public const short MinValue = -32768;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static short Parse(string s);
   public static short Parse(string s, NumberStyles style);
   public static short Parse(string s, IFormatProvider provider);
   public static short Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public struct Int32: IComparable, IFormattable
```

```
{
   public const int MaxValue = 2147483647;
   public const int MinValue = -2147483648;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static int Parse(string s);
   public static int Parse(string s, NumberStyles style);
   public static int Parse(string s, IFormatProvider provider);
   public static int Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public struct Int64: IComparable, IFormattable
{
   public const long MaxValue = 9223372036854775807;
   public const long MinValue = -9223372036854775808;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static long Parse(string s);
   public static long Parse(string s, NumberStyles style);
   public static long Parse(string s, IFormatProvider provider);
   public static long Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System.Threading, Library: BCL
public sealed class Interlocked
{
   public static int CompareExchange(ref int location1, int value, int
      comparand);
   public static float CompareExchange(ref float location1, float value, float
      comparand);
   public static object CompareExchange(ref object location1, object value,
```

```
      object comparand);
   public static int Decrement(ref int location);
   public static long Decrement(ref long location);
   public static int Exchange(ref int location1, int value);
   public static float Exchange(ref float location1, float value);
   public static object Exchange(ref object location1, object value);
   public static int Increment(ref int location);
   public static long Increment(ref long location);
}

// Namespace: System, Library: BCL
public class InvalidCastException: SystemException
{
   public InvalidCastException();
   public InvalidCastException(string message);
   public InvalidCastException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public class InvalidOperationException: SystemException
{
   public InvalidOperationException();
   public InvalidOperationException(string message);
   public InvalidOperationException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public sealed class InvalidProgramException: SystemException
{
   public InvalidProgramException();
   public InvalidProgramException(string message);
   public InvalidProgramException(string message, Exception inner);
}

// Namespace: System.IO, Library: BCL
public class IOException: SystemException
{
   public IOException();
   public IOException(string message);
   public IOException(string message, Exception innerException);
}

// Namespace: System.Security, Library: BCL
public interface IPermission
```

```
{
   IPermission Copy();
   void Demand();
   IPermission Intersect(IPermission target);
   bool IsSubsetOf(IPermission target);
   IPermission Union(IPermission target);
}

// Namespace: System, Library: BCL
public abstract class MarshalByRefObject
{
}

// Namespace: System, Library: ExtendedNumerics
public sealed class Math
{
   public const double E = 2.71828182845905;
   public const double PI = 3.14159265358979;
   public static sbyte Abs(sbyte value);
   public static short Abs(short value);
   public static int Abs(int value);
   public static long Abs(long value);
   public static float Abs(float value);
   public static double Abs(double value);
   public static decimal Abs(decimal value);
   public static double Acos(double d);
   public static double Asin(double d);
   public static double Atan(double d);
   public static double Atan2(double y, double x);
   public static long BigMul(int a, int b);
   public static double Ceiling(double a);
   public static double Cos(double d);
   public static double Cosh(double value);
   public static int DivRem(int a, int b, out int result);
   public static long DivRem(long a, long b, out long result);
   public static double Exp(double d);
   public static double Floor(double d);
   public static double IEEERemainder(double x, double y);
   public static double Log(double d);
   public static double Log(double a, double newBase);
   public static double Log10(double d);
   public static sbyte Max(sbyte val1, sbyte val2);
   public static byte Max(byte val1, byte val2);
   public static short Max(short val1, short val2);
```

```
    public static ushort Max(ushort val1, ushort val2);
    public static int Max(int val1, int val2);
    public static uint Max(uint val1, uint val2);
    public static long Max(long val1, long val2);
    public static ulong Max(ulong val1, ulong val2);
    public static float Max(float val1, float val2);
    public static double Max(double val1, double val2);
    public static decimal Max(decimal val1, decimal val2);
    public static sbyte Min(sbyte val1, sbyte val2);
    public static byte Min(byte val1, byte val2);
    public static short Min(short val1, short val2);
    public static ushort Min(ushort val1, ushort val2);
    public static int Min(int val1, int val2);
    public static uint Min(uint val1, uint val2);
    public static long Min(long val1, long val2);
    public static ulong Min(ulong val1, ulong val2);
    public static float Min(float val1, float val2);
    public static double Min(double val1, double val2);
    public static decimal Min(decimal val1, decimal val2);
    public static double Pow(double x, double y);
    public static double Round(double a);
    public static double Round(double value, int digits);
    public static decimal Round(decimal d);
    public static int Sign(sbyte value);
    public static int Sign(short value);
    public static int Sign(int value);
    public static int Sign(long value);
    public static int Sign(float value);
    public static int Sign(double value);
    public static int Sign(decimal value);
    public static double Sin(double a);
    public static double Sinh(double value);
    public static double Sqrt(double d);
    public static double Tan(double a);
    public static double Tanh(double value);
}

// Namespace: System.IO, Library: BCL
public class MemoryStream: Stream
{
    public MemoryStream();
    public MemoryStream(int capacity);
    public MemoryStream(byte[] buffer);
    public MemoryStream(byte[] buffer, bool writable);
```

```
    public MemoryStream(byte[] buffer, int index, int count);
    public MemoryStream(byte[] buffer, int index, int count, bool writable);
    public MemoryStream(byte[] buffer, int index, int count, bool writable,
       bool publiclyVisible);
    public override void Close();
    public override void Flush();
    public virtual byte[] GetBuffer();
    public override int Read(byte[] buffer, int offset, int count);
    public override int ReadByte();
    public override long Seek(long offset, SeekOrigin loc);
    public override void SetLength(long value);
    public virtual byte[] ToArray();
    public override void Write(byte[] buffer, int offset, int count);
    public override void WriteByte(byte value);
    public virtual void WriteTo(Stream stream);
    public override bool CanRead { get; }
    public override bool CanSeek { get; }
    public override bool CanWrite { get; }
    public virtual int Capacity { get; set; }
    public override long Length { get; }
    public override long Position { get; set; }
}

// Namespace: System.Threading, Library: BCL
public sealed class Monitor
{
    public static void Enter(object obj);
    public static void Exit(object obj);
    public static void Pulse(object obj);
    public static void PulseAll(object obj);
    public static bool TryEnter(object obj);
    public static bool TryEnter(object obj, int millisecondsTimeout);
    public static bool TryEnter(object obj, TimeSpan timeout);
    public static bool Wait(object obj, int millisecondsTimeout);
    public static bool Wait(object obj, TimeSpan timeout);
    public static bool Wait(object obj);
}

// Namespace: System, Library: ExtendedNumerics
public class NotFiniteNumberException: ArithmeticException
{
    public NotFiniteNumberException();
    public NotFiniteNumberException(double offendingNumber);
    public NotFiniteNumberException(string message);
```

```
    public NotFiniteNumberException(string message, double offendingNumber);
    public NotFiniteNumberException(string message, double offendingNumber,
       Exception innerException);
    public double OffendingNumber { get; }
}

// Namespace: System, Library: BCL
public class NotSupportedException: SystemException
{
    public NotSupportedException();
    public NotSupportedException(string message);
    public NotSupportedException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public class NullReferenceException: SystemException
{
    public NullReferenceException();
    public NullReferenceException(string message);
    public NullReferenceException(string message, Exception innerException);
}

// Namespace: System.Globalization, Library: BCL
public sealed class NumberFormatInfo: ICloneable, IFormatProvider
{
    public NumberFormatInfo();
    public object Clone();
    public object GetFormat(Type formatType);
    public static NumberFormatInfo ReadOnly(NumberFormatInfo nfi);
    public int CurrencyDecimalDigits { get; set; }
    public string CurrencyDecimalSeparator { get; set; }
    public string CurrencyGroupSeparator { get; set; }
    public int[] CurrencyGroupSizes { get; set; }
    public int CurrencyNegativePattern { get; set; }
    public int CurrencyPositivePattern { get; set; }
    public string CurrencySymbol { get; set; }
    public static NumberFormatInfo CurrentInfo { get; }
    public static NumberFormatInfo InvariantInfo { get; }
    public bool IsReadOnly { get; }
    public string NaNSymbol { get; set; }
    public string NegativeInfinitySymbol { get; set; }
    public string NegativeSign { get; set; }
    public int NumberDecimalDigits { get; set; }
    public string NumberDecimalSeparator { get; set; }
```

```
    public string NumberGroupSeparator { get; set; }
    public int[] NumberGroupSizes { get; set; }
    public int NumberNegativePattern { get; set; }
    public int PercentDecimalDigits { get; set; }
    public string PercentDecimalSeparator { get; set; }
    public string PercentGroupSeparator { get; set; }
    public int[] PercentGroupSizes { get; set; }
    public int PercentNegativePattern { get; set; }
    public int PercentPositivePattern { get; set; }
    public string PercentSymbol { get; set; }
    public string PerMilleSymbol { get; set; }
    public string PositiveInfinitySymbol { get; set; }
    public string PositiveSign { get; set; }
}

// Namespace: System.Globalization, Library: BCL
public enum NumberStyles
{
    AllowCurrencySymbol = 0x100,
    AllowDecimalPoint = 0x20,
    AllowExponent = 0x80,
    AllowHexSpecifier = 0x200,
    AllowLeadingSign = 0x4,
    AllowLeadingWhite = 0x1,
    AllowParentheses = 0x10,
    AllowThousands = 0x40,
    AllowTrailingSign = 0x8,
    AllowTrailingWhite = 0x2,
    Any = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
        AllowTrailingSign | AllowParentheses | AllowDecimalPoint |
        AllowThousands | AllowExponent | AllowCurrencySymbol,
    Currency = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
        AllowTrailingSign | AllowParentheses | AllowDecimalPoint |
        AllowThousands | AllowCurrencySymbol,
    Float = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
        AllowDecimalPoint | AllowExponent,
    HexNumber = AllowLeadingWhite | AllowTrailingWhite | AllowHexSpecifier,
    Integer = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign,
    None = 0x0,
    Number = AllowLeadingWhite | AllowTrailingWhite | AllowLeadingSign |
        AllowTrailingSign | AllowDecimalPoint | AllowThousands,
}

// Namespace: System, Library: BCL
```

```
public class Object
{
   public Object();
   public virtual bool Equals(object obj);
   public static bool Equals(object objA, object objB);
   ~Object();
   public virtual int GetHashCode();
   public Type GetType();
   protected object MemberwiseClone();
   public static bool ReferenceEquals(object objA, object objB);
   public virtual string ToString();
}

// Namespace: System, Library: BCL
public class ObjectDisposedException: InvalidOperationException
{
   public ObjectDisposedException(string objectName);
   public ObjectDisposedException(string objectName, string message);
   public override string Message { get; }
   public string ObjectName { get; }
}

// Namespace: System, Library: BCL
public sealed class ObsoleteAttribute: Attribute
{
   public ObsoleteAttribute();
   public ObsoleteAttribute(string message);
   public ObsoleteAttribute(string message, bool error);
   public bool IsError { get; }
   public string Message { get; }
}

// Namespace: System, Library: BCL
public class OutOfMemoryException: SystemException
{
   public OutOfMemoryException();
   public OutOfMemoryException(string message);
   public OutOfMemoryException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public class OverflowException: ArithmeticException
{
   public OverflowException();
```

```
   public OverflowException(string message);
   public OverflowException(string message, Exception innerException);
}

// Namespace: System.IO, Library: BCL
public sealed class Path
{
   public static readonly char AltDirectorySeparatorChar;
   public static readonly char DirectorySeparatorChar;
   public static readonly char PathSeparator;
   public static string ChangeExtension(string path, string extension);
   public static string Combine(string path1, string path2);
   public static string GetDirectoryName(string path);
   public static string GetExtension(string path);
   public static string GetFileName(string path);
   public static string GetFileNameWithoutExtension(string path);
   public static string GetFullPath(string path);
   public static string GetPathRoot(string path);
   public static string GetTempFileName();
   public static string GetTempPath();
   public static bool HasExtension(string path);
   public static bool IsPathRooted(string path);
}

// Namespace: System.IO, Library: BCL
public class PathTooLongException: IOException
{
   public PathTooLongException();
   public PathTooLongException(string message);
   public PathTooLongException(string message, Exception innerException);
}

// Namespace: System.Security, Library: BCL
public class PermissionSet: ICollection, IEnumerable
{
   public PermissionSet(PermissionState state);
   public PermissionSet(PermissionSet permSet);
   public virtual IPermission AddPermission(IPermission perm);
   public virtual void Assert();
   public virtual PermissionSet Copy();
   public virtual void CopyTo(Array array, int index);
   public virtual void Demand();
   public virtual void Deny();
   public virtual void FromXml(SecurityElement et);
```

```
    public virtual IEnumerator GetEnumerator();
    public virtual bool IsSubsetOf(PermissionSet target);
    public virtual void PermitOnly();
    public override string ToString();
    public virtual SecurityElement ToXml();
    public virtual PermissionSet Union(PermissionSet other);
    int ICollection.Count { get; }
    bool ICollection.IsSynchronized { get; }
    object ICollection.SyncRoot { get; }
}

// Namespace: System.Security.Permissions, Library: BCL
public enum PermissionState
{
    None = 0,
    Unrestricted = 1,
}

// Namespace: System, Library: BCL
public class Random
{
    public Random();
    public Random(int Seed);
    public virtual int Next(int maxValue);
    public virtual int Next(int minValue, int maxValue);
    public virtual int Next();
    public virtual void NextBytes(byte[] buffer);
    public virtual double NextDouble();
}

// Namespace: System, Library: BCL
public class RankException: SystemException
{
    public RankException();
    public RankException(string message);
    public RankException(string message, Exception innerException);
}

// Namespace: System, Library: BCL
public struct SByte: IComparable, IFormattable
{
    public const sbyte MaxValue = 127;
    public const sbyte MinValue = -128;
    public int CompareTo(object obj);
```

```
    public override bool Equals(object obj);
    public override int GetHashCode();
    public static sbyte Parse(string s);
    public static sbyte Parse(string s, NumberStyles style);
    public static sbyte Parse(string s, IFormatProvider provider);
    public static sbyte Parse(string s, NumberStyles style, IFormatProvider
       provider);
    public string ToString(IFormatProvider provider);
    public string ToString(string format, IFormatProvider provider);
    public override string ToString();
    public string ToString(string format);
}

// Namespace: System.Security.Permissions, Library: BCL
public enum SecurityAction
{
    Assert = 3,
    Demand = 2,
    Deny = 4,
    InheritanceDemand = 7,
    LinkDemand = 6,
    PermitOnly = 5,
    RequestMinimum = 8,
    RequestOptional = 9,
    RequestRefuse = 10,
}

// Namespace: System.Security.Permissions, Library: BCL
public abstract class SecurityAttribute: Attribute
{
    protected SecurityAttribute();
    public SecurityAttribute(SecurityAction action);
    public abstract IPermission CreatePermission();
    public bool Unrestricted { get; set; }
}

// Namespace: System.Security, Library: BCL
public sealed class SecurityElement
{
    public override string ToString();
}

// Namespace: System.Security, Library: BCL
public class SecurityException: SystemException
```

```
{
   public SecurityException();
   public SecurityException(string message);
   public SecurityException(string message, Exception inner);
}

// Namespace: System.Security.Permissions, Library: BCL
public sealed class SecurityPermission: CodeAccessPermission
{
   public SecurityPermission(PermissionState state);
   public SecurityPermission(SecurityPermissionFlag flag);
   public override IPermission Copy();
   public override void FromXml(SecurityElement esd);
   public override IPermission Intersect(IPermission target);
   public override bool IsSubsetOf(IPermission target);
   public override SecurityElement ToXml();
   public override IPermission Union(IPermission target);
}

// Namespace: System.Security.Permissions, Library: BCL
public sealed class SecurityPermissionAttribute: CodeAccessSecurityAttribute
{
   public SecurityPermissionAttribute(SecurityAction action);
   public override IPermission CreatePermission();
   public SecurityPermissionFlag Flags { get; set; }
}

// Namespace: System.Security.Permissions, Library: BCL
public enum SecurityPermissionFlag
{
   Assertion = 0x1,
   ControlThread = 0x10,
   Execution = 0x8,
   NoFlags = 0x0,
   SkipVerification = 0x4,
   UnmanagedCode = 0x2,
}

// Namespace: System.IO, Library: BCL
public enum SeekOrigin
{
   Begin = 0,
   Current = 1,
   End = 2,
```

```
}

// Namespace: System, Library: ExtendedNumerics
public struct Single: IComparable, IFormattable
{
   public const float Epsilon = (float)1.401298E-45;
   public const float MaxValue = (float)3.402823E+38;
   public const float MinValue = (float)-3.402823E+38;
   public const float NaN = (float)0.0 / (float)0.0;
   public const float NegativeInfinity = (float)-1.0 / (float)0.0;
   public const float PositiveInfinity = (float)1.0 / (float)0.0;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static bool IsInfinity(float f);
   public static bool IsNaN(float f);
   public static bool IsNegativeInfinity(float f);
   public static bool IsPositiveInfinity(float f);
   public static float Parse(string s);
   public static float Parse(string s, NumberStyles style);
   public static float Parse(string s, IFormatProvider provider);
   public static float Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public sealed class StackOverflowException: SystemException
{
   public StackOverflowException();
   public StackOverflowException(string message);
   public StackOverflowException(string message, Exception innerException);
}

// Namespace: System.IO, Library: BCL
public abstract class Stream: MarshalByRefObject, IDisposable
{
   protected Stream();
   public static readonly Stream Null;
   public virtual IAsyncResult BeginRead(byte[] buffer, int offset, int count,
      AsyncCallback callback, object state);
```

```
    public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int
       count, AsyncCallback callback, object state);
    public virtual void Close();
    protected virtual WaitHandle CreateWaitHandle();
    public virtual int EndRead(IAsyncResult asyncResult);
    public virtual void EndWrite(IAsyncResult asyncResult);
    public abstract void Flush();
    public abstract int Read(byte[] buffer, int offset, int count);
    public virtual int ReadByte();
    public abstract long Seek(long offset, SeekOrigin origin);
    public abstract void SetLength(long value);
    void IDisposable.Dispose();
    public abstract void Write(byte[] buffer, int offset, int count);
    public virtual void WriteByte(byte value);
    public abstract bool CanRead { get; }
    public abstract bool CanSeek { get; }
    public abstract bool CanWrite { get; }
    public abstract long Length { get; }
    public abstract long Position { get; set; }
}

// Namespace: System.IO, Library: BCL
public class StreamReader: TextReader
{
    public StreamReader(Stream stream);
    public StreamReader(Stream stream, bool detectEncodingFromByteOrderMarks);
    public StreamReader(Stream stream, Encoding encoding);
    public StreamReader(Stream stream, Encoding encoding, bool
       detectEncodingFromByteOrderMarks);
    public StreamReader(Stream stream, Encoding encoding, bool
       detectEncodingFromByteOrderMarks, int bufferSize);
    public StreamReader(string path);
    public StreamReader(string path, bool detectEncodingFromByteOrderMarks);
    public StreamReader(string path, Encoding encoding);
    public StreamReader(string path, Encoding encoding, bool
       detectEncodingFromByteOrderMarks);
    public StreamReader(string path, Encoding encoding, bool
       detectEncodingFromByteOrderMarks, int bufferSize);
    public override void Close();
    public void DiscardBufferedData();
    protected override void Dispose(bool disposing);
    public override int Peek();
    public override int Read(char[] buffer, int index, int count);
    public override int Read();
```

```
    public override string ReadLine();
    public override string ReadToEnd();
    public virtual Stream BaseStream { get; }
    public virtual Encoding CurrentEncoding { get; }
}

// Namespace: System.IO, Library: BCL
public class StreamWriter: TextWriter
{
    public StreamWriter(Stream stream);
    public StreamWriter(Stream stream, Encoding encoding);
    public StreamWriter(Stream stream, Encoding encoding, int bufferSize);
    public StreamWriter(string path);
    public StreamWriter(string path, bool append);
    public StreamWriter(string path, bool append, Encoding encoding);
    public StreamWriter(string path, bool append, Encoding encoding, int
        bufferSize);
    public override void Close();
    protected override void Dispose(bool disposing);
    ~StreamWriter();
    public override void Flush();
    public override void Write(string value);
    public override void Write(char[] buffer, int index, int count);
    public override void Write(char[] buffer);
    public override void Write(char value);
    public virtual bool AutoFlush { get; set; }
    public virtual Stream BaseStream { get; }
    public override Encoding Encoding { get; }
}

// Namespace: System, Library: BCL
public sealed class String: ICloneable, IComparable, IEnumerable
{
    unsafe public String(char* value);
    unsafe public String(char* value, int startIndex, int length);
    public String(char[] value, int startIndex, int length);
    public String(char[] value);
    public String(char c, int count);
    public static readonly string Empty;
    public object Clone();
    public static int Compare(string strA, string strB);
    public static int Compare(string strA, string strB, bool ignoreCase);
    public static int Compare(string strA, int indexA, string strB, int indexB,
        int length);
```

```
    public static int Compare(string strA, int indexA, string strB, int indexB,
       int length, bool ignoreCase);
    public static int CompareOrdinal(string strA, string strB);
    public static int CompareOrdinal(string strA, int indexA, string strB, int
       indexB, int length);
    public int CompareTo(object value);
    public static string Concat(object arg0, object arg1);
    public static string Concat(object arg0, object arg1, object arg2);
    public static string Concat(params object[] args);
    public static string Concat(string str0, string str1);
    public static string Concat(string str0, string str1, string str2);
    public static string Concat(params string[] values);
    public static string Copy(string str);
    public void CopyTo(int sourceIndex, char[] destination, int
       destinationIndex, int count);
    public bool EndsWith(string value);
    public override bool Equals(object obj);
    public static bool Equals(string a, string b);
    public static string Format(string format, object arg0);
    public static string Format(string format, object arg0, object arg1);
    public static string Format(string format, object arg0, object arg1, object
       arg2);
    public static string Format(string format, params object[] args);
    public static string Format(IFormatProvider provider, string format, params
       object[] args);
    public CharEnumerator GetEnumerator();
    public override int GetHashCode();
    public int IndexOf(char value);
    public int IndexOf(char value, int startIndex);
    public int IndexOf(char value, int startIndex, int count);
    public int IndexOf(string value);
    public int IndexOf(string value, int startIndex);
    public int IndexOf(string value, int startIndex, int count);
    public int IndexOfAny(char[] anyOf);
    public int IndexOfAny(char[] anyOf, int startIndex);
    public int IndexOfAny(char[] anyOf, int startIndex, int count);
    public string Insert(int startIndex, string value);
    public static string Intern(string str);
    public static string IsInterned(string str);
    public static string Join(string separator, string[] value);
    public static string Join(string separator, string[] value, int startIndex,
       int count);
    public int LastIndexOf(char value);
    public int LastIndexOf(char value, int startIndex);
```

```
    public int LastIndexOf(char value, int startIndex, int count);
    public int LastIndexOf(string value);
    public int LastIndexOf(string value, int startIndex);
    public int LastIndexOf(string value, int startIndex, int count);
    public int LastIndexOfAny(char[] anyOf);
    public int LastIndexOfAny(char[] anyOf, int startIndex);
    public int LastIndexOfAny(char[] anyOf, int startIndex, int count);
    public static bool operator ==(String a, String b);
    public static bool operator !=(String a, String b);
    public string PadLeft(int totalWidth);
    public string PadLeft(int totalWidth, char paddingChar);
    public string PadRight(int totalWidth);
    public string PadRight(int totalWidth, char paddingChar);
    public string Remove(int startIndex, int count);
    public string Replace(char oldChar, char newChar);
    public string Replace(string oldValue, string newValue);
    public string[] Split(params char[] separator);
    public string[] Split(char[] separator, int count);
    public bool StartsWith(string value);
    public string Substring(int startIndex);
    public string Substring(int startIndex, int length);
    IEnumerator IEnumerable.GetEnumerator();
    public char[] ToCharArray();
    public char[] ToCharArray(int startIndex, int length);
    public string ToLower();
    public string ToString(IFormatProvider provider);
    public override string ToString();
    public string ToUpper();
    public string Trim(params char[] trimChars);
    public string Trim();
    public string TrimEnd(params char[] trimChars);
    public string TrimStart(params char[] trimChars);
    public char this[int index] { get; }
    public int Length { get; }
}

// Namespace: System.Text, Library: BCL
public sealed class StringBuilder
{
    public StringBuilder();
    public StringBuilder(int capacity);
    public StringBuilder(string value);
    public StringBuilder Append(char value, int repeatCount);
    public StringBuilder Append(char[] value, int startIndex, int charCount);
```

```
public StringBuilder Append(string value);
public StringBuilder Append(string value, int startIndex, int count);
public StringBuilder Append(bool value);
public StringBuilder Append(sbyte value);
public StringBuilder Append(byte value);
public StringBuilder Append(char value);
public StringBuilder Append(short value);
public StringBuilder Append(int value);
public StringBuilder Append(long value);
public StringBuilder Append(float value);
public StringBuilder Append(double value);
public StringBuilder Append(decimal value);
public StringBuilder Append(ushort value);
public StringBuilder Append(uint value);
public StringBuilder Append(ulong value);
public StringBuilder Append(object value);
public StringBuilder Append(char[] value);
public StringBuilder AppendFormat(string format, object arg0);
public StringBuilder AppendFormat(string format, object arg0, object arg1);
public StringBuilder AppendFormat(string format, object arg0, object arg1,
   object arg2);
public StringBuilder AppendFormat(string format, params object[] args);
public StringBuilder AppendFormat(IFormatProvider provider, string format,
   params object[] args);
public int EnsureCapacity(int capacity);
public bool Equals(StringBuilder sb);
public StringBuilder Insert(int index, string value, int count);
public StringBuilder Insert(int index, string value);
public StringBuilder Insert(int index, bool value);
public StringBuilder Insert(int index, sbyte value);
public StringBuilder Insert(int index, byte value);
public StringBuilder Insert(int index, short value);
public StringBuilder Insert(int index, char value);
public StringBuilder Insert(int index, char[] value);
public StringBuilder Insert(int index, char[] value, int startIndex, int
   charCount);
public StringBuilder Insert(int index, int value);
public StringBuilder Insert(int index, long value);
public StringBuilder Insert(int index, float value);
public StringBuilder Insert(int index, double value);
public StringBuilder Insert(int index, decimal value);
public StringBuilder Insert(int index, ushort value);
public StringBuilder Insert(int index, uint value);
public StringBuilder Insert(int index, ulong value);
```

```
    public StringBuilder Insert(int index, object value);
    public StringBuilder Remove(int startIndex, int length);
    public StringBuilder Replace(string oldValue, string newValue);
    public StringBuilder Replace(string oldValue, string newValue, int
       startIndex, int count);
    public StringBuilder Replace(char oldChar, char newChar);
    public StringBuilder Replace(char oldChar, char newChar, int startIndex,
       int count);
    public override string ToString();
    public string ToString(int startIndex, int length);
    public int Capacity { get; set; }
    public char this[int index] { get; set; }
    public int Length { get; set; }
}

// Namespace: System.IO, Library: BCL
public class StringReader: TextReader
{
    public StringReader(string s);
    public override void Close();
    protected override void Dispose(bool disposing);
    public override int Peek();
    public override int Read(char[] buffer, int index, int count);
    public override int Read();
    public override string ReadLine();
    public override string ReadToEnd();
}

// Namespace: System.IO, Library: BCL
public class StringWriter: TextWriter
{
    public StringWriter();
    public StringWriter(IFormatProvider formatProvider);
    public StringWriter(StringBuilder sb);
    public StringWriter(StringBuilder sb, IFormatProvider formatProvider);
    public override void Close();
    protected override void Dispose(bool disposing);
    public virtual StringBuilder GetStringBuilder();
    public override string ToString();
    public override void Write(string value);
    public override void Write(char[] buffer, int index, int count);
    public override void Write(char value);
    public override Encoding Encoding { get; }
}
```

```
// Namespace: System.Threading, Library: BCL
public class SynchronizationLockException: SystemException
{
   public SynchronizationLockException();
   public SynchronizationLockException(string message);
   public SynchronizationLockException(string message, Exception
      innerException);
}

// Namespace: System, Library: BCL
public class SystemException: Exception
{
   public SystemException();
   public SystemException(string message);
   public SystemException(string message, Exception innerException);
}

// Namespace: System.IO, Library: BCL
public abstract class TextReader: MarshalByRefObject, IDisposable
{
   protected TextReader();
   public static readonly TextReader Null;
   public virtual void Close();
   protected virtual void Dispose(bool disposing);
   public virtual int Peek();
   public virtual int Read(char[] buffer, int index, int count);
   public virtual int Read();
   public virtual int ReadBlock(char[] buffer, int index, int count);
   public virtual string ReadLine();
   public virtual string ReadToEnd();
   public static TextReader Synchronized(TextReader reader);
   void IDisposable.Dispose();
}

// Namespace: System.IO, Library: BCL
public abstract class TextWriter: MarshalByRefObject, IDisposable
{
   protected TextWriter();
   protected TextWriter(IFormatProvider formatProvider);
   public static readonly TextWriter Null;
   public virtual void Close();
   protected virtual void Dispose(bool disposing);
   public virtual void Flush();
```

```
    public static TextWriter Synchronized(TextWriter writer);
    void IDisposable.Dispose();
    public virtual void Write(string format, params object[] arg);
    public virtual void Write(string format, object arg0, object arg1, object
       arg2);
    public virtual void Write(string format, object arg0, object arg1);
    public virtual void Write(string format, object arg0);
    public virtual void Write(object value);
    public virtual void Write(string value);
    public virtual void Write(decimal value);
    public virtual void Write(double value);
    public virtual void Write(float value);
    public virtual void Write(ulong value);
    public virtual void Write(long value);
    public virtual void Write(uint value);
    public virtual void Write(int value);
    public virtual void Write(bool value);
    public virtual void Write(char[] buffer, int index, int count);
    public virtual void Write(char[] buffer);
    public virtual void Write(char value);
    public virtual void WriteLine(string format, params object[] arg);
    public virtual void WriteLine(string format, object arg0, object arg1,
       object arg2);
    public virtual void WriteLine(string format, object arg0, object arg1);
    public virtual void WriteLine(string format, object arg0);
    public virtual void WriteLine(object value);
    public virtual void WriteLine(string value);
    public virtual void WriteLine(decimal value);
    public virtual void WriteLine(double value);
    public virtual void WriteLine(float value);
    public virtual void WriteLine(ulong value);
    public virtual void WriteLine(long value);
    public virtual void WriteLine(uint value);
    public virtual void WriteLine(int value);
    public virtual void WriteLine(bool value);
    public virtual void WriteLine(char[] buffer, int index, int count);
    public virtual void WriteLine(char[] buffer);
    public virtual void WriteLine(char value);
    public virtual void WriteLine();
    public abstract Encoding Encoding { get; }
    public virtual IFormatProvider FormatProvider { get; }
    public virtual string NewLine { get; set; }
}
```

```
// Namespace: System, Library: BCL
public sealed class Thread
{
   public Thread(ThreadStart start);
   public void Abort(object stateInfo);
   public void Abort();
   ~Thread();
   public void Join();
   public bool Join(int millisecondsTimeout);
   public bool Join(TimeSpan timeout);
   public static void MemoryBarrier ();
   public static void ResetAbort();
   public static void Sleep(int millisecondsTimeout);
   public static void Sleep(TimeSpan timeout);
   public void Start();
   public static byte VolatileRead (ref byte address);
   public static short VolatileRead (ref short address);
   public static int VolatileRead (ref int address);
   public static long VolatileRead (ref long address);
   public static sbyte VolatileRead (ref sbyte address);
   public static ushort VolatileRead (ref ushort address);
   public static uint VolatileRead (ref uint address);
   public static ulong VolatileRead (ref ulong address);
   public static float VolatileRead (ref float address);
   public static double VolatileRead (ref double address);
   public static object VolatileRead (ref object address);
   public static void VolatileWrite (ref byte address, byte value);
   public static void VolatileWrite (ref short address, short value);
   public static void VolatileWrite (ref int address, int value);
   public static void VolatileWrite (ref long address, long value);
   public static void VolatileWrite (ref sbyte address, sbyte value);
   public static void VolatileWrite (ref ushort address, ushort value);
   public static void VolatileWrite (ref uint address, uint value);
   public static void VolatileWrite (ref ulong address, ulong value);
   public static void VolatileWrite (ref float address, float value);
   public static void VolatileWrite (ref double address, double value);
   public static void VolatileWrite (ref object address, object value);
   public static Thread CurrentThread { get; }
   public bool IsAlive { get; }
   public bool IsBackground { get; set; }
   public string Name { get; set; }
   public ThreadPriority Priority { get; set; }
   public ThreadState ThreadState { get; }
}
```

```
// Namespace: System.Threading, Library: BCL
public sealed class ThreadAbortException: SystemException
{
   public object ExceptionState { get; }
}

// Namespace: System.Threading, Library: BCL
public enum ThreadPriority
{
   AboveNormal = 3,
   BelowNormal = 1,
   Highest = 4,
   Lowest = 0,
   Normal = 2,
}

// Namespace: System.Threading, Library: BCL
public delegate void ThreadStart();

// Namespace: System.Threading, Library: BCL
public enum ThreadState
{
   Aborted = 0x100,
   AbortRequested = 0x80,
   Background = 0x4,
   Running = 0x0,
   Stopped = 0x10,
   Unstarted = 0x8,
   WaitSleepJoin = 0x20,
}

// Namespace: System.Threading, Library: BCL
public class ThreadStateException: SystemException
{
   public ThreadStateException();
   public ThreadStateException(string message);
   public ThreadStateException(string message, Exception innerException);
}

// Namespace: System.Threading, Library: BCL
public sealed class Timeout
{
   public const int Infinite = -1;
```

```
}

// Namespace: System.Threading, Library: BCL
public sealed class Timer: MarshalByRefObject, IDisposable
{
   public Timer(TimerCallback callback, object state, int dueTime, int period);
   public Timer(TimerCallback callback, object state, TimeSpan dueTime,
      TimeSpan period);
   public bool Change(int dueTime, int period);
   public bool Change(TimeSpan dueTime, TimeSpan period);
   public void Dispose();
   public bool Dispose(WaitHandle notifyObject);
   ~Timer();
}

// Namespace: System.Threading, Library: BCL
public delegate void TimerCallback(object state);

// Namespace: System, Library: BCL
public struct TimeSpan: IComparable
{
   public TimeSpan(long ticks);
   public TimeSpan(int hours, int minutes, int seconds);
   public TimeSpan(int days, int hours, int minutes, int seconds);
   public TimeSpan(int days, int hours, int minutes, int seconds, int
      milliseconds);
   public static readonly TimeSpan MaxValue;
   public static readonly TimeSpan MinValue;
   public const long TicksPerDay = 864000000000;
   public const long TicksPerHour = 36000000000;
   public const long TicksPerMillisecond = 10000;
   public const long TicksPerMinute = 600000000;
   public const long TicksPerSecond = 10000000;
   public static readonly TimeSpan Zero;
   public TimeSpan Add(TimeSpan ts);
   public static int Compare(TimeSpan t1, TimeSpan t2);
   public int CompareTo(object value);
   public TimeSpan Duration();
   public override bool Equals(object value);
   public static bool Equals(TimeSpan t1, TimeSpan t2);
   public static TimeSpan FromDays(double value);
   public static TimeSpan FromHours(double value);
   public static TimeSpan FromMilliseconds(double value);
   public static TimeSpan FromMinutes(double value);
```

```
    public static TimeSpan FromSeconds(double value);
    public static TimeSpan FromTicks(long value);
    public override int GetHashCode();
    public TimeSpan Negate();
    public static TimeSpan operator +(TimeSpan t1, TimeSpan t2);
    public static bool operator ==(TimeSpan t1, TimeSpan t2);
    public static bool operator >(TimeSpan t1, TimeSpan t2);
    public static bool operator >=(TimeSpan t1, TimeSpan t2);
    public static bool operator !=(TimeSpan t1, TimeSpan t2);
    public static bool operator  <(TimeSpan t1, TimeSpan t2);
    public static bool operator <=(TimeSpan t1, TimeSpan t2);
    public static TimeSpan operator -(TimeSpan t1, TimeSpan t2);
    public static TimeSpan operator -(TimeSpan t);
    public static TimeSpan operator +(TimeSpan t);
    public static TimeSpan Parse(string s);
    public TimeSpan Subtract(TimeSpan ts);
    public override string ToString();
    public int Days { get; }
    public int Hours { get; }
    public int Milliseconds { get; }
    public int Minutes { get; }
    public int Seconds { get; }
    public long Ticks { get; }
    public double TotalDays { get; }
    public double TotalHours { get; }
    public double TotalMilliseconds { get; }
    public double TotalMinutes { get; }
    public double TotalSeconds { get; }
}

// Namespace: System, Library: BCL
public abstract class Type: Object
{
    public virtual int GetArrayRank();
    public abstract Type GetElementType();
    public override int GetHashCode();
    public virtual bool IsAssignableFrom(Type c);
    public virtual bool IsInstanceOfType(object o);
    public virtual bool IsSubclassOf(Type c);
    public override string ToString();
    public abstract Type BaseType { get; }
    public abstract string FullName { get; }
    public bool IsArray { get; }
    public bool IsClass { get; }
```

```
   public bool IsEnum { get; }
   public bool IsInterface { get; }
   public bool IsPointer { get; }
   public bool IsValueType { get; }
}

// Namespace: System, Library: BCL
public sealed class TypeInitializationException: SystemException
{
   public string TypeName { get; }
}

// Namespace: System, Library: BCL
public struct UInt16: IComparable, IFormattable
{
   public const ushort MaxValue = 65535;
   public const ushort MinValue = 0;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static ushort Parse(string s);
   public static ushort Parse(string s, NumberStyles style);
   public static ushort Parse(string s, IFormatProvider provider);
   public static ushort Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public struct UInt32: IComparable, IFormattable
{
   public const uint MaxValue = 4294967295;
   public const uint MinValue = 0;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static uint Parse(string s);
   public static uint Parse(string s, NumberStyles style);
   public static uint Parse(string s, IFormatProvider provider);
   public static uint Parse(string s, NumberStyles style, IFormatProvider
      provider);
```

```
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public struct UInt64: IComparable, IFormattable
{
   public const ulong MaxValue = 18446744073709551615;
   public const ulong MinValue = 0;
   public int CompareTo(object value);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static ulong Parse(string s);
   public static ulong Parse(string s, NumberStyles style);
   public static ulong Parse(string s, IFormatProvider provider);
   public static ulong Parse(string s, NumberStyles style, IFormatProvider
      provider);
   public string ToString(IFormatProvider provider);
   public string ToString(string format, IFormatProvider provider);
   public override string ToString();
   public string ToString(string format);
}

// Namespace: System, Library: BCL
public class UnauthorizedAccessException: SystemException
{
   public UnauthorizedAccessException();
   public UnauthorizedAccessException(string message);
   public UnauthorizedAccessException(string message, Exception inner);
}

// Namespace: System.Globalization, Library: BCL
public enum UnicodeCategory
{
   ClosePunctuation = 21,
   ConnectorPunctuation = 18,
   Control = 14,
   CurrencySymbol = 26,
   DashPunctuation = 19,
   DecimalDigitNumber = 8,
   EnclosingMark = 7,
   FinalQuotePunctuation = 23,
```

```
    Format = 15,
    InitialQuotePunctuation = 22,
    LetterNumber = 9,
    LineSeparator = 12,
    LowercaseLetter = 1,
    MathSymbol = 25,
    ModifierLetter = 3,
    ModifierSymbol = 27,
    NonSpacingMark = 5,
    OpenPunctuation = 20,
    OtherLetter = 4,
    OtherNotAssigned = 29,
    OtherNumber = 10,
    OtherPunctuation = 24,
    OtherSymbol = 28,
    ParagraphSeparator = 13,
    PrivateUse = 17,
    SpaceSeparator = 11,
    SpacingCombiningMark = 6,
    Surrogate = 16,
    TitlecaseLetter = 2,
    UppercaseLetter = 0,
}

// Namespace: System.Text, Library: BCL
public class UnicodeEncoding: Encoding
{
    public UnicodeEncoding();
    public UnicodeEncoding(bool bigEndian, bool byteOrderMark);
    public override bool Equals(object value);
    public override int GetByteCount(char[] chars, int index, int count);
    public override int GetByteCount(string s);
    public override int GetBytes(string s, int charIndex, int charCount, byte[]
        bytes, int byteIndex);
    public override byte[] GetBytes(string s);
    public override int GetBytes(char[] chars, int charIndex, int charCount,
        byte[] bytes, int byteIndex);
    public override int GetCharCount(byte[] bytes, int index, int count);
    public override int GetChars(byte[] bytes, int byteIndex, int byteCount,
        char[] chars, int charIndex);
    public override Decoder GetDecoder();
    public override int GetHashCode();
    public override int GetMaxByteCount(int charCount);
    public override int GetMaxCharCount(int byteCount);
```

```
   public override byte[] GetPreamble();
}

// Namespace: System.Text, Library: BCL
public class UTF8Encoding: Encoding
{
   public UTF8Encoding();
   public UTF8Encoding(bool encoderShouldEmitUTF8Identifier);
   public UTF8Encoding(bool encoderShouldEmitUTF8Identifier, bool
      throwOnInvalidBytes);
   public override bool Equals(object value);
   public override int GetByteCount(char[] chars, int index, int count);
   public override int GetByteCount(string chars);
   public override int GetBytes(string s, int charIndex, int charCount, byte[]
      bytes, int byteIndex);
   public override byte[] GetBytes(string s);
   public override int GetBytes(char[] chars, int charIndex, int charCount,
      byte[] bytes, int byteIndex);
   public override int GetCharCount(byte[] bytes, int index, int count);
   public override int GetChars(byte[] bytes, int byteIndex, int byteCount,
      char[] chars, int charIndex);
   public override Decoder GetDecoder();
   public override Encoder GetEncoder();
   public override int GetHashCode();
   public override int GetMaxByteCount(int charCount);
   public override int GetMaxCharCount(int byteCount);
   public override byte[] GetPreamble();
}

// Namespace: System, Library: BCL
public abstract class ValueType
{
   protected ValueType();
   public override bool Equals(object obj);
   public override int GetHashCode();
   public override string ToString();
}

// Namespace: System.Security, Library: BCL
public class VerificationException: SystemException
{
   public VerificationException();
   public VerificationException(string message);
   public VerificationException(string message, Exception innerException);
```

```
}

// Namespace: System, Library: BCL
public sealed class Version: ICloneable, IComparable
{
   public Version(int major, int minor, int build, int revision);
   public Version(int major, int minor, int build);
   public Version(int major, int minor);
   public Version(string version);
   public Version();
   public object Clone();
   public int CompareTo(object version);
   public override bool Equals(object obj);
   public override int GetHashCode();
   public static bool operator ==(Version v1, Version v2);
   public static bool operator >(Version v1, Version v2);
   public static bool operator >=(Version v1, Version v2);
   public static bool operator !=(Version v1, Version v2);
   public static bool operator  <(Version v1, Version v2);
   public static bool operator <=(Version v1, Version v2);
   public int Build { get; }
   public int Major { get; }
   public int Minor { get; }
   public int Revision { get; }
}

// Namespace: System.Threading, Library: BCL
public abstract class WaitHandle: MarshalByRefObject, IDisposable
{
   public WaitHandle();
   public virtual void Close();
   protected virtual void Dispose(bool explicitDisposing);
   ~WaitHandle();
   void IDisposable.Dispose();
   public static bool WaitAll(WaitHandle[] waitHandles);
   public static int WaitAny(WaitHandle[] waitHandles);
   public virtual bool WaitOne();
}
```

End of informative text.

E. Documentation Comments

This clause is informative.

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. Comments using such syntax are called ***documentation comments***. The XML generation tool is called the ***documentation generator***. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the ***documentation file***. A documentation file is used as input to a ***documentation viewer***; a tool intended to produce some sort of visual display of type information and its associated documentation.

A conforming C# compiler is not required to check the syntax of documentation comments; such comments are simply ordinary comments. A conforming compiler is permitted to do such checking, however.

This specification suggests a set of standard tags to be used in documentation comments. For C# implementations targeting the CLI, it also provides information about the documentation generator and the format of the documentation file. No information is provided about the documentation viewer.

E.1 Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are *single-line comment*s of the form `///…` or *delimited* comments of the form `/** … */`. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate.

Syntax:

single-line-doc-comment::
`///` *input-characters*$_{opt}$

delimited-doc-comment::
`/**` *delimited-comment-characters*$_{opt}$ `*/`

In a *single-line-doc-comment*, if there is a *whitespace* character following the `///` characters on each of the *single-line-doc-comments* adjacent to the current *single-line-doc-comment*, then that *whitespace* character is not included in the XML output.

In a *delimited-doc-comment*, if the first non-*whitespace* character on the second line is an *asterisk* and the same pattern of optional *whitespace* characters and an *asterisk* character is repeated at the beginning of each of the lines within the *delimited-doc-comment*, then the characters of the repeated pattern are not included in the XML output. The pattern may include *whitespace* characters after, as well as before, the *asterisk* character.

Example:

```
/**
 * <remarks>Class <c>Point</c> models a point in a two-dimensional
 * plane.</remarks>
 */
public class Point
{
   /// <remarks>method <c>draw</c> renders the point.</remarks>
   void draw() {…}
}
```

The text within documentation comments must be well formed according to the rules of XML (http://www.w3.org/TR/REC-xml). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in §E.2. Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a `cref` attribute, the documentation generator must respect namespace visibility according to `using` statements appearing within the source code.
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

E.2 Recommended tags

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

Tag	Reference	Purpose
`<c>`	§E.2.1	Set text in a code-like font
`<code>`	§E.2.2	Set one or more lines of source code or program output
`<example>`	§E.2.3	Indicate an example
`<exception>`	§E.2.4	Identifies the exceptions a method can throw
`<list>`	§E.2.5	Create a list or table
`<para>`	§E.2.6	Permit structure to be added to text
`<param>`	§E.2.7	Describe a parameter for a method or constructor
`<paramref>`	§E.2.8	Identify that a word is a parameter name
`<permission>`	§E.2.9	Document the security accessibility of a member
`<remarks>`	§E.2.10	Describe a type
`<returns>`	§E.2.11	Describe the return value of a method
`<see>`	§E.2.12	Specify a link
`<seealso>`	§E.2.13	Generate a *See Also* entry
`<summary>`	§E.2.14	Describe a member of a type
`<value>`	§E.2.15	Describe a property

E.2.1 <c>

This tag provides a mechanism to indicate that a fragment of text within a description should be set a special font such as that used for a block of code. (For lines of actual code, use `<code>` (§E.2.2).)

Syntax:

<c>*text to be set like code*</c>

Example:

```
/// <remarks>Class <c>Point</c> models a point in a two-dimensional
/// plane.</remarks>
public class Point
{
   // …
}
```

E.2.2 <code>

This tag is used to set one or more lines of source code or program output in some special font. (For small code fragments in narrative, use <c> (§E.2.1).)

Syntax:

<code>*source code or program output*</code>

Example:

```
/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// <example>For example:
/// <code>
///    Point p = new Point(3,5);
///    p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>

public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

E.2.3 <example>

This tag allows example code within a comment, to specify how a method or other library member may be used. Ordinarily, this would also involve use of the tag <code> (§E.2.2) as well.

Syntax:

<example>*description*</example>

Example:

See <code> (§E.2.2) for an example.

E.2.4 <exception>

This tag provides a way to document the exceptions a method can throw.

Syntax:

<exception cref="*member*">*description*</exception>

where

cref="*member*"

The name of a member. The documentation generator checks that the given member exists and translates *member* to the canonical element name in the documentation file.

description

A description of the circumstances in which the exception is thrown.

Example:

```
public class DataBaseOperations
{
   /// <exception cref="MasterFileFormatCorruptException"></exception>
   /// <exception cref="MasterFileLockedOpenException"></exception>
   public static void ReadRecord(int flag) {
      if (flag == 1)
         throw new MasterFileFormatCorruptException();
      else if (flag == 2)
         throw new MasterFileLockedOpenException();
      // …
   }
}
```

E.2.5 <list>

This tag is used to create a list or table of items. It may contain a `<listheader>` block to define the heading row of either a table or definition list. (When defining a table, only an entry for *term* in the heading need be supplied.)

Each item in the list is specified with an `<item>` block. When creating a definition list, both *term* and *description* must be specified. However, for a table, bulleted list, or numbered list, only *description* need be specified.

Syntax:

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
   …
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

where

term

The term to define, whose definition is in *description*.

description

Either an item in a bullet or numbered list, or the definition of a *term*.

Example:

```
public class MyClass
{
   /// <remarks>Here is an example of a bulleted list:
   /// <list type="bullet">
   /// <item>
   /// <description>Item 1.</description>
   /// </item>
   /// <item>
   /// <description>Item 2.</description>
   /// </item>
   /// </list>
   /// </remarks>
   public static void Main () {
      // …
   }
}
```

E.2.6 <para>

This tag is for use inside other tags, such as <remarks> (§E.2.10) or <returns> (§E.2.11), and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

where

content

The text of the paragraph.

Example:

```
/// <summary>This is the entry point of the Point class testing
program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trvial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
   // …
}
```

E.2.7 <param>

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

name

The name of the parameter.

description

A description of the parameter.

Example:

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <param><c>xor</c> is the new x-coordinate.</param>
/// <param><c>yor</c> is the new y-coordinate.</param>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

E.2.8 <paramref>

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

where

name

The name of the parameter.

Example:

```
        /// <summary>This constructor initializes the new Point to
        ///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
        /// <param><c>xor</c> is the new Point's x-coordinate.</param>
        /// <param><c>yor</c> is the new Point's y-coordinate.</param>
        public Point(int xor, int yor) {
           X = xor;
           Y = yor;
        }
```

E.2.9 <permission>

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

`cref="member"`

The name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.

description

A description of the access to the member.

Example:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>
public static void Test() {
   // …
}
```

E.2.10 <remarks>

This tag is used to specify overview information about a type. (Use `<summary>` (§E.2.14) to describe the members of a type.)

Syntax:

```
<remarks>description</remarks>
```

where

description

The text of the remarks.

Example:

```
/// <remarks>Class <c>Point</c> models a point in a two-dimensional
plane.</remarks>
public class Point
{
   // …
}
```

E.2.11 <returns>

This tag is used to describe the return value of a method.

Syntax:

```
<returns>description</returns>
```

where

description

A description of the return value.

Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form
(x,y),
///   without any leading, training, or embedded whitespace.</returns>
public override string ToString() {
   return "(" + X + "," + Y + ")";
}
```

E.2.12 <see>

This tag allows a link to be specified within text. (Use `<seealso>` (§E.2.13) to indicate text that is to appear in a *See Also* section.)

Syntax:

```
<see cref="member"/>
```

where

cref="*member*"

The name of a member. The documentation generator checks that the given code element exists and passes *member* to the element name in the documentation file.

Example:

```
/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}

/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

E.2.13 <seealso>

This tag allows an entry to be generated for the *See Also* section. (Use `<see>` (§E.2.12) to specify a link from within text.)

Syntax:

```
<seealso cref="member"/>
```

where

cref="*member*"

The name of a member. The documentation generator checks that the given code element exists and passes *member* to the element name in the documentation file.

Example:

```
/// <summary>This method determines whether two Points have the same
///   location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
   // …
}
```

E.2.14 <summary>

This tag can be used to describe a member for a type. (Use `<remarks>` (§E.2.10) to describe the type itself.)

Syntax:

```
<summary>description</summary>
```

where

description

A summary of the member.

Example:

```
/// <summary>This constructor initializes the new Point to
(0,0).</summary>
public Point() : this(0,0) {
}
```

E.2.15 <value>

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where

property description

A description for the property.

Example:

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
   get { return x; }
   set { x = value; }
}
```

E.3 Processing the documentation file

The following information is intended for C# implementations targeting the CLI.

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding metadata/reflection item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

E.3.1 ID string format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined:

Character	Description
E	Event
F	Field
M	Method (including constructors, destructors, and operators)
N	Namespace
P	Property (including indexers)
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the NUMBER SIGN `# (U+000D)`. (It is assumed that no element has this character in its name.)
- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows: Arguments are represented by their fully qualified name. For example, `int` becomes `System.Int32`, `string` becomes `System.String`, `object` becomes `System.Object`, and so on. Arguments having the `out` or `ref` modifier have a '@' following their type name. Arguments passed by value or via `params` have no special notation. Arguments that are arrays are represented as [*lowerbound*:*size*, …, *lowerbound*:*size*] where the number of commas is the rank – 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. Jagged arrays are represented by one "[]" per level. Arguments that have pointer types other than void are represented using a '*' following the type name. A void pointer is represented using a type name of `"System.Void"`.

E.3.2 ID string examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name.

```
enum Color {Red, Blue, Green};
namespace Acme
{
interface IProcess { /* … */ }
struct ValueType { /* … */ }

class Widget: Iprocess
{
   public class NestedClass { /* … */ }
   public interface IMenuItem { /* … */ }
   public delegate void Del(int i);
   public enum Direction {North, South, East, West};
}
}
```

```
"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
```

- Fields are represented by their fully qualified name.

```
namespace Acme
{
struct ValueType
{
   private int total;
}
class Widget: Iprocess
{
   public class NestedClass
   {
      private int value;
   }
   private string message;
   private static Color defaultColor;
   private const double PI = 3.14159;
   protected readonly double monthlyAverage;
   private long[] array1;
   private Widget[,] array2;
   private unsafe int *pCount;
   private unsafe float **ppValues;
}
}
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Constructors.

```
namespace Acme
{
class Widget: Iprocess
{
   static Widget() { /* … */ }
   public Widget() { /* … */ }
   public Widget(string s) { /* … */ }
}
}
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Destructors.

```
namespace Acme
{
class Widget: Iprocess
{
   ~Widget() { /* … */ }
}
}
```

```
"M:Acme.Widget.Finalize"
```

- Methods.

```
namespace Acme
{
struct ValueType
{
   public void M(int i) { /* … */ }
}

class Widget: Iprocess
{
   public class NestedClass
   {
      public void M(int i) { /* … */ }
   }
   public static void M0() { /* … */ }
   public void M1(char c, out float f, ref ValueType v) { /* … */ }
   public void M2(short[] x1, int[,] x2, long[][] x3) { /* … */ }
   public void M3(long[][] x3, Widget[][,,] x4) { /* … */ }
   public unsafe void M4(char *pc, Color **pf) { /* … */ }
   public unsafe void M5(void *pv, double *[][,] pd) { /* … */ }
   public void M6(int i, params object[] args) { /* … */ }
}
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(=VOID*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
```

- Properties and indexers.

```
namespace Acme
{
class Widget: Iprocess
{
   public int Width {get { /* … */ } set { /* … */ }}
   public int this[int i] {get { /* … */ } set { /* … */ }}
   public int this[string s, int i] {get { /* … */ } set { /* … */ }}
}
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- Events

```
namespace Acme
{
class Widget: Iprocess
{
   public event Del AnEvent;
}
}

"E:Acme.Widget.AnEvent"
```

- Unary operators.

```
namespace Acme
{
class Widget: Iprocess
{
   public static Widget operator+(Widget x) { /* … */ }
}
}
```

```
"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

The complete set of unary operator function names used is as follows: `op_UnaryPlus`, `op_UnaryNegation`, `op_Negation`, `op_OnesComplement`, `op_Increment`, `op_Decrement`, `op_True`, and `op_False`.

- Binary operators.

```
namespace Acme
{
class Widget: Iprocess
{
   public static Widget operator+(Widget x1, Widget x2) { return x1; }
}
}
```

```
"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

The complete set of binary operator function names used is as follows: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus`, `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_Equality`, `op_Inequality`, `op_LessThan`, `op_LessThanOrEqual`, `op_GreaterThan`, and `op_GreaterThanOrEqual`.

- Conversion operators have a trailing '~' followed by the return type.

```
namespace Acme
{
class Widget: Iprocess
{
   public static explicit operator int(Widget x) { /* … */ }
   public static implicit operator long(Widget x) { /* … */ }
}
}
```

```
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

E.4 An example

E.4.1 C# source code

The following example shows the source code of a Point class:

```
namespace Graphics
{

/// <remarks>Class <c>Point</c> models a point in a two-dimensional
plane.
/// </remarks>
public class Point
{
   /// <summary>Instance variable <c>x</c> represents the point's
   ///   x-coordinate.</summary>
   private int x;

   /// <summary>Instance variable <c>y</c> represents the point's
   ///   y-coordinate.</summary>
   private int y;
```

```
   /// <value>Property <c>X</c> represents the point's x-
coordinate.</value>
   public int X
   {
      get { return x; }
      set { x = value; }
   }
   /// <value>Property <c>Y</c> represents the point's y-
coordinate.</value>
   public int Y
   {
      get { return y; }
      set { y = value; }
   }
   /// <summary>This constructor initializes the new Point to
   ///    (0,0).</summary>
   public Point() : this(0,0) {}
   /// <summary>This constructor initializes the new Point to
   ///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
   /// <param><c>xor</c> is the new Point's x-coordinate.</param>
   /// <param><c>yor</c> is the new Point's y-coordinate.</param>
   public Point(int xor, int yor) {
      X = xor;
      Y = yor;
   }
   /// <summary>This method changes the point's location to
   ///    the given coordinates.</summary>
   /// <param><c>xor</c> is the new x-coordinate.</param>
   /// <param><c>yor</c> is the new y-coordinate.</param>
   /// <see cref="Translate"/>
   public void Move(int xor, int yor) {
      X = xor;
      Y = yor;
   }
   /// <summary>This method changes the point's location by
   ///    the given x- and y-offsets.
   /// <example>For example:
   /// <code>
   ///    Point p = new Point(3,5);
   ///    p.Translate(-1,3);
   /// </code>
   /// results in <c>p</c>'s having the value (2,8).
   /// </example>
   /// </summary>
   /// <param><c>xor</c> is the relative x-offset.</param>
   /// <param><c>yor</c> is the relative y-offset.</param>
   /// <see cref="Move"/>
   public void Translate(int xor, int yor) {
      X += xor;
      Y += yor;
   }
   /// <summary>This method determines whether two Points have the same
   ///    location.</summary>
   /// <param><c>o</c> is the object to be compared to the current
object.
   /// </param>
   /// <returns>True if the Points have the same location and they have
   ///    the exact same type; otherwise, false.</returns>
   /// <seealso cref="operator=="/>
   /// <seealso cref="operator!="/>
   public override bool Equals(object o) {
      if (o == null) {
         return false;
      }
```

```
      if (this == o) {
         return true;
      }
      if (GetType() == o.GetType()) {
         Point p = (Point)o;
         return (X == p.X) && (Y == p.Y);
      }
      return false;
   }
   /// <summary>Report a point's location as a string.</summary>
   /// <returns>A string representing a point's location, in the form
(x,y),
   ///   without any leading, training, or embedded whitespace.</returns>
   public override string ToString() {
      return "(" + X + "," + Y + ")";
   }
   /// <summary>This operator determines whether two Points have the same
   ///   location.</summary>
   /// <param><c>p1</c> is the first Point to be compared.</param>
   /// <param><c>p2</c> is the second Point to be compared.</param>
   /// <returns>True if the Points have the same location and they have
   ///   the exact same type; otherwise, false.</returns>
   /// <seealso cref="Equals"/>
   /// <seealso cref="operator!="/>
   public static bool operator==(Point p1, Point p2) {
      if ((object)p1 == null || (object)p2 == null) {
         return false;
      }

      if (p1.GetType() == p2.GetType()) {
         return (p1.X == p2.X) && (p1.Y == p2.Y);
      }

      return false;
   }
   /// <summary>This operator determines whether two Points have the same
   ///   location.</summary>
   /// <param><c>p1</c> is the first Point to be compared.</param>
   /// <param><c>p2</c> is the second Point to be compared.</param>
   /// <returns>True if the Points do not have the same location and the
   ///   exact same type; otherwise, false.</returns>
   /// <seealso cref="Equals"/>
   /// <seealso cref="operator=="/>
   public static bool operator!=(Point p1, Point p2) {
      return !(p1 == p2);
   }
   /// <summary>This is the entry point of the Point class testing
   /// program.
   /// <para>This program tests each method and operator, and
   /// is intended to be run after any non-trvial maintenance has
   /// been performed on the Point class.</para></summary>
   public static void Main() {
      // class test code goes here
   }
}
}
```

E.4.2 Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <remarks>Class <c>Point</c> models a point in a two-
dimensional
            plane.
            </remarks>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>

        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>

        <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
            <summary>This method changes the point's location to
            the given coordinates.</summary>
            <param><c>xor</c> is the new x-coordinate.</param>
            <param><c>yor</c> is the new y-coordinate.</param>
            <see
cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
        </member>

        <member
            name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
            <summary>This method changes the point's location by
            the given x- and y-offsets.
            <example>For example:
            <code>
            Point p = new Point(3,5);
            p.Translate(-1,3);
            </code>
            results in <c>p</c>'s having the value (2,8).
            </example>
            </summary>
            <param><c>xor</c> is the relative x-offset.</param>
            <param><c>yor</c> is the relative y-offset.</param>
            <see
cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
        </member>
```

```
            <member name="M:Graphics.Point.Equals(System.Object)">
                <summary>This method determines whether two Points have the
    same
                location.</summary>
                <param><c>o</c> is the object to be compared to the current
                object.
                </param>
                <returns>True if the Points have the same location and they
    have
                the exact same type; otherwise, false.</returns>
                <seealso

    cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
                <seealso

    cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
            </member>
            <member name="M:Graphics.Point.ToString">
                <summary>Report a point's location as a string.</summary>
                <returns>A string representing a point's location, in the
    form
                (x,y),
                without any leading, training, or embedded
    whitespace.</returns>
            </member>
            <member

    name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
                <summary>This operator determines whether two Points have the
                same
                location.</summary>
                <param><c>p1</c> is the first Point to be compared.</param>
                <param><c>p2</c> is the second Point to be compared.</param>
                <returns>True if the Points have the same location and they
    have
                the exact same type; otherwise, false.</returns>
                <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
                <seealso

    cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
            </member>
            <member

    name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
                <summary>This operator determines whether two Points have the
                same
                location.</summary>
                <param><c>p1</c> is the first Point to be compared.</param>
                <param><c>p2</c> is the second Point to be compared.</param>
                <returns>True if the Points do not have the same location and
                the
                exact same type; otherwise, false.</returns>
                <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
                <seealso

    cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
            </member>
            <member name="M:Graphics.Point.Main">
                <summary>This is the entry point of the Point class testing
                program.
                <para>This program tests each method and operator, and
                is intended to be run after any non-trvial maintenance has
                been performed on the Point class.</para></summary>
            </member>
```

```
            <member name="P:Graphics.Point.X">
                <value>Property <c>X</c> represents the point's
                x-coordinate.</value>
            </member>

            <member name="P:Graphics.Point.Y">
                <value>Property <c>Y</c> represents the point's
                y-coordinate.</value>
            </member>
        </members>
    </doc>
```

F. Index

This clause is informative.

1

1

Free printed copies can be ordered from:
ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Email: documents@ecma.ch

Files of this Standard can be freely downloaded from the ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.