Contents

C# 6.0 draft language specification

5/24/2018 • 2 minutes to read • Edit Online

The C# language specification is the definitive source for C# syntax and usage. This specification contains detailed information about all aspects of the language, including many points that the documentation for C# doesn't cover.

Version 5.0 of the specification has been released in December 2017 as the Standard ECMA-334 5th Edition document.

Version 6.0 of the specification has not been approved as a standard. This site contains the *draft* C# 6.0 specification. It's built from the markdown files contained in the dotnet/csharplang GitHub repository.

Issues on the draft specification should be created in the dotnet/csharplang repository. Or, if you are interested in fixing any errors you find, you may submit a Pull Request to the same repository.

See also

C# Reference
C# Programming Guide

NEXT

Introduction

1/13/2018 • 55 minutes to read • Edit Online

C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers. C# is standardized by ECMA International as the ***ECMA-334*** standard and by ISO/IEC as the ***ISO/IEC 23270*** standard. Microsoft's C# compiler for the .NET Framework is a conforming implementation of both of these standards.

C# is an object-oriented language, but C# further includes support for ***component-oriented*** programming. Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to directly support these concepts, making C# a very natural language in which to create and use software components.

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

C# has a ***unified type system***. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. Thus, all types share a set of common operations, and values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

To ensure that C# programs and libraries can evolve over time in a compatible manner, much emphasis has been placed on ***versioning*** in C#'s design. Many programming languages pay little attention to this issue, and, as a result, programs written in those languages break more often than necessary when newer versions of dependent libraries are introduced. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

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

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

C# source files typically have the file extension `.cs`. Assuming that the "Hello, World" program is stored in the file `hello.cs`, the program can be compiled with the Microsoft C# compiler using the command line

```
csc hello.cs
```

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

```
Hello, World
```

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the `static` modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the .NET Framework class libraries, which, by default, are automatically referenced by the Microsoft C# compiler. Note that C# itself does not have a separate runtime library. Instead, the .NET Framework is the runtime library of C#.

Program structure

The key organizational concepts in C# are ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***. C# programs consist of one or more source files. Programs declare types, which contain members and can be organized into namespaces. Classes and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they are physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement ***applications*** or ***libraries***.

The example

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

declares a class named `Stack` in a namespace called `Acme.Collections`. The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. Assuming that the source code of the example is stored in the file `acme.cs`, the command line

```
csc /t:library acme.cs
```

compiles the example as a library (code without a `Main` entry point) and produces an assembly named `acme.dll`.

Assemblies contain executable code in the form of ***Intermediate Language*** (IL) instructions, and symbolic information in the form of ***metadata***. Before it is executed, the IL code in an assembly is automatically converted to processor-specific code by the Just-In-Time (JIT) compiler of .NET Common Language Runtime.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for `#include` directives and header files in C#. The public types and members contained in a particular assembly are made available in a C# program simply by referencing that assembly when compiling the program. For example, this program uses the `Acme.Collections.Stack` class from the `acme.dll` assembly:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

If the program is stored in the file `test.cs`, when `test.cs` is compiled, the `acme.dll` assembly can be referenced using the compiler's `/r` option:

```
csc /r:acme.dll test.cs
```

This creates an executable assembly named `test.exe`, which, when run, produces the output:

```
100
10
1
```

C# permits the source text of a program to be stored in several source files. When a multi-file C# program is compiled, all of the source files are processed together, and the source files can freely reference each other—conceptually, it is as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with very few exceptions, declaration order is insignificant. C# does not limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

Types and variables

There are two kinds of types in C#: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and 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 (except in the case of `ref` and `out` parameter variables).

C#'s value types are further divided into ***simple types***, ***enum types***, ***struct types***, and ***nullable types***, and C#'s reference types are further divided into ***class types***, ***interface types***, ***array types***, and ***delegate types***.

The following table provides an overview of C#'s type system.

CATEGORY		DESCRIPTION
Value types	Simple types	Signed integral: `sbyte`, `short`, `int`, `long`
		Unsigned integral: `byte`, `ushort`, `uint`, `ulong`

CATEGORY		DESCRIPTION
		Unicode characters: `char`
		IEEE floating point: `float`, `double`
		High-precision decimal: `decimal`
		Boolean: `bool`
	Enum types	User-defined types of the form `enum E {...}`
	Struct types	User-defined types of the form `struct S {...}`
	Nullable types	Extensions of all other value types with a `null` value
Reference types	Class types	Ultimate base class of all other types: `object`
		Unicode strings: `string`
		User-defined types of the form `class C {...}`
	Interface types	User-defined types of the form `interface I {...}`
	Array types	Single- and multi-dimensional, for example, `int[]` and `int[,]`
	Delegate types	User-defined types of the form e.g. `delegate int D(...)`

The eight integral types provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

The two floating point types, `float` and `double`, are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats.

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations.

C#'s `bool` type is used to represent boolean values—values that are either `true` or `false`.

Character and string processing in C# uses Unicode encoding. The `char` type represents a UTF-16 code unit, and the `string` type represents a sequence of UTF-16 code units.

The following table summarizes C#'s numeric types.

CATEGORY	BITS	TYPE	RANGE/PRECISION
Signed integral	8	`sbyte`	-128...127

CATEGORY	BITS	TYPE	RANGE/PRECISION
	16	`short`	-32,768...32,767
	32	`int`	-2,147,483,648...2,147,483,647
	64	`long`	-9,223,372,036,854,775,808...9,223,372,036,854,775,807
Unsigned integral	8	`byte`	0...255
	16	`ushort`	0...65,535
	32	`uint`	0...4,294,967,295
	64	`ulong`	0...18,446,744,073,709,551,615
Floating point	32	`float`	1.5 × 10^−45 to 3.4 × 10^38, 7-digit precision
	64	`double`	5.0 × 10^−324 to 1.7 × 10^308, 15-digit precision
Decimal	128	`decimal`	1.0 × 10^−28 to 7.9 × 10^28, 28-digit precision

C# programs use ***type declarations*** to create new types. A type declaration specifies the name and the members of the new type. Five of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, and delegate types.

A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

A struct type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and do not require heap allocation. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

An interface type defines a contract as a named set of public function members. A class or struct that implements an interface must provide implementations of the interface's function members. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

Class, struct, interface and delegate types all support generics, whereby they can be parameterized with other types.

An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying

type.

C# supports single- and multi-dimensional arrays of any type. Unlike the types listed above, array types do not have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays of `int`.

Nullable types also do not have to be declared before they can be used. For each non-nullable value type `T` there is a corresponding nullable type `T?`, which can hold an additional value `null`. For instance, `int?` is a type that can hold any 32 bit integer or the value `null`.

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

```
using System;

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

When a value of a value type is converted to type `object`, an object instance, also called a "box," is allocated to hold the value, and the value is copied into that box. Conversely, when an `object` reference is cast to a value type, a check is made that the referenced object is a box of the correct value type, and, if the check succeeds, the value in the box is copied out.

C#'s unified type system effectively means that value types can become objects "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with both reference types and value types.

There are several kinds of ***variables*** in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations, and every variable has a type that determines what values can be stored in the variable, as shown by the following table.

TYPE OF VARIABLE	POSSIBLE CONTENTS
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
`object`	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type
Class type	A null reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
Interface type	A null reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type

TYPE OF VARIABLE	POSSIBLE CONTENTS
Array type	A null reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
Delegate type	A null reference or a reference to an instance of that delegate type

Expressions

Expressions are constructed from ***operands*** and ***operators***. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

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

Most operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

The following table summarizes C#'s operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

CATEGORY	EXPRESSION	DESCRIPTION
Primary	`x.m`	Member access
	`x(...)`	Method and delegate invocation
	`x[...]`	Array and indexer access
	`x++`	Post-increment
	`x--`	Post-decrement
	`new T(...)`	Object and delegate creation
	`new T(...){...}`	Object creation with initializer
	`new {...}`	Anonymous object initializer
	`new T[...]`	Array creation
	`typeof(T)`	Obtain `System.Type` object for `T`
	`checked(x)`	Evaluate expression in checked context
	`unchecked(x)`	Evaluate expression in unchecked context
	`default(T)`	Obtain default value of type `T`

CATEGORY	EXPRESSION	DESCRIPTION
	`delegate {...}`	Anonymous function (anonymous method)
Unary	`+x`	Identity
	`-x`	Negation
	`!x`	Logical negation
	`~x`	Bitwise negation
	`++x`	Pre-increment
	`--x`	Pre-decrement
	`(T)x`	Explicitly convert `x` to type `T`
	`await x`	Asynchronously wait for `x` to complete
Multiplicative	`x * y`	Multiplication
	`x / y`	Division
	`x % y`	Remainder
Additive	`x + y`	Addition, string concatenation, delegate combination
	`x - y`	Subtraction, delegate removal
Shift	`x << y`	Shift left
	`x >> y`	Shift right
Relational and type testing	`x < y`	Less than
	`x > y`	Greater than
	`x <= y`	Less than or equal
	`x >= y`	Greater than or equal
	`x is T`	Return `true` if `x` is a `T`, `false` otherwise
	`x as T`	Return `x` typed as `T`, or `null` if `x` is not a `T`

CATEGORY	EXPRESSION	DESCRIPTION
Equality	`x == y`	Equal
	`x != y`	Not equal
Logical AND	`x & y`	Integer bitwise AND, boolean logical AND
Logical XOR	`x ^ y`	Integer bitwise XOR, boolean logical XOR
Logical OR	`x | y`	Integer bitwise OR, boolean logical OR
Conditional AND	`x && y`	Evaluates `y` only if `x` is `true`
Conditional OR	`x || y`	Evaluates `y` only if `x` is `false`
Null coalescing	`X ?? y`	Evaluates to `y` if `x` is `null`, to `x` otherwise
Conditional	`x ? y : z`	Evaluates `y` if `x` is `true`, `z` if `x` is `false`
Assignment or anonymous function	`x = y`	Assignment
	`x op= y`	Compound assignment; supported operators are `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`
	`(T x) => y`	Anonymous function (lambda expression)

Statements

The actions of a program are expressed using ***statements***. C# supports several different kinds of statements, a number of which are defined in terms of embedded statements.

A ***block*** permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.

Declaration statements are used to declare local variables and constants.

Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound assignment operators, increment and decrement operations using the `++` and `--` operators and await expressions.

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the `if` and `switch` statements.

Iteration statements are used to repeatedly execute an embedded statement. In this group are the `while`, `do`, `for`, and `foreach` statements.

Jump statements are used to transfer control. In this group are the `break`, `continue`, `goto`, `throw`, `return`, and

`yield` statements.

The `try` ... `catch` statement is used to catch exceptions that occur during execution of a block, and the `try` ... `finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.

The `checked` and `unchecked` statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.

The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

Below are examples of each kind of statement

Local variable declarations

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

Local constant declaration

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

Expression statement

```
static void Main() {
    int i;
    i = 123;               // Expression statement
    Console.WriteLine(i);  // Expression statement
    i++;                   // Expression statement
    Console.WriteLine(i);  // Expression statement
}
```

`if` statement

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

`switch` statement

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

`while` statement

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

`do` statement

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

`for` statement

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

`foreach` statement

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

`break` statement

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

`continue` statement

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

`goto` statement

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

`return` statement

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

`yield` statement

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

`throw` and `try` statements

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

`checked` and `unchecked` statements

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

`lock` statement

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

`using` statement

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

Classes and objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created ***instances*** of the class, also known as ***objects***. Classes support ***inheritance*** and ***polymorphism***, mechanisms whereby ***derived classes*** can extend and specialize ***base classes***.

New classes are created using class declarations. A class declaration starts with a header that specifies the attributes and modifiers of the class, the name of the class, the base class (if given), and the interfaces implemented by the class. The header is followed by the class body, which consists of a list of member declarations written between the delimiters `{` and `}`.

The following is a declaration of a simple class named `Point`:

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

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

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in C#.

Members

The members of a class are either ***static members*** or ***instance members***. Static members belong to classes, and instance members belong to objects (instances of classes).

The following table provides an overview of the kinds of members a class can contain.

MEMBER	DESCRIPTION
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class

MEMBER	DESCRIPTION
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, 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 following table.

ACCESSIBILITY	MEANING
`public`	Access not limited
`protected`	Access limited to this class or classes derived from this class
`internal`	Access limited to this program
`protected internal`	Access limited to this program or classes derived from this class
`private`	Access limited to this class

Type parameters

A class definition may specify a set of type parameters by following the class name with angle brackets enclosing a list of type parameter names. The type parameters can the be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond`:

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

A class type that is declared to take type parameters is called a generic class type. Struct, interface and delegate types can also be generic.

When the generic class is used, type arguments must be provided for each of the type parameters:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

A generic type with type arguments provided, like `Pair<int,string>` above, is called a constructed type.

Base classes

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object`. In the following example, the base class of `Point3D` is `Point`, and the base class of `Point` is `object`:

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

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

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains all members of its base class, except for the instance and static constructors, and the destructors of the base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member. In the previous example, `Point3D` inherits the `x` and `y` fields from `Point`, and every `Point3D` instance contains three fields, `x`, `y`, and `z`.

An implicit conversion exists from a class type to any of its base class types. Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Fields

A field is a variable that is associated with a class or with an instance of a class.

A field declared with the `static` modifier defines a ***static field***. A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.

A field declared without the `static` modifier defines an ***instance field***. Every instance of a class contains a separate copy of all the instance fields of that class.

In the following example, each instance of the `Color` class has a separate copy of the `r`, `g`, and `b` instance fields, but there is only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

As shown in the previous example, ***read-only fields*** may be declared with a `readonly` modifier. Assignment to a `readonly` field can only occur as part of the field's declaration or in a constructor in the same class.

Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class. ***Static methods*** are accessed through the class. ***Instance methods*** are accessed through instances of the class.

Methods have a (possibly empty) list of ***parameters***, which represent values or variable references passed to the method, and a ***return type***, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it does not return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The ***signature*** of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters and the number, modifiers, and types of its parameters. The signature of a method does not include the return type.

Parameters

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the ***arguments*** that are specified when the method is invoked. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A ***value parameter*** is used for input parameter passing. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed for the parameter.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

A ***reference parameter*** is used for both input and output parameter passing. The argument passed for a reference parameter must be a variable, and during execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);            // Outputs "2 1"
    }
}
```

An ***output parameter*** is used for output parameter passing. An output parameter is similar to a reference parameter except that the initial value of the caller-provided argument is unimportant. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters.

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

A ***parameter array*** permits a variable number of arguments to be passed to a method. A parameter array is declared with the `params` modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. The `Write` and `WriteLine` methods of the `System.Console` class are good examples of parameter array usage. They are declared as follows.

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Within a method that uses a parameter array, the parameter array behaves exactly like a regular parameter of an array type. However, in an invocation of a method with a parameter array, it is possible to pass either a single argument of the parameter array type or any number of arguments of the element type of the parameter array. In the latter case, an array instance is automatically created and initialized with the given arguments. This example

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

is equivalent to writing the following.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Method body and local variables

A method's body specifies the statements to execute when the method is invoked.

A method body can declare variables that are specific to the invocation of the method. Such variables are called ***local variables***. A local variable declaration specifies a type name, a variable name, and possibly an initial value. The following example declares a local variable `i` with an initial value of zero and a local variable `j` with no initial value.

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requires a local variable to be ***definitely assigned*** before its value can be obtained. For example, if the declaration of the previous `i` did not include an initial value, the compiler would report an error for the subsequent usages of `i` because `i` would not be definitely assigned at those points in the program.

A method can use `return` statements to return control to its caller. In a method returning `void`, `return` statements cannot specify an expression. In a method returning non-`void`, `return` statements must include an expression that computes the return value.

Static and instance methods

A method declared with a `static` modifier is a ***static method***. A static method does not operate on a specific instance and can only directly access static members.

A method declared without a `static` modifier is an ***instance method***. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as `this`. It is an error to refer to `this` in a static method.

The following `Entity` class has both static and instance members.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

Each `Entity` instance contains a serial number (and presumably some other information that is not shown here). The `Entity` constructor (which is like an instance method) initializes the new instance with the next available serial number. Because the constructor is an instance member, it is permitted to access both the `serialNo` instance field and the `nextSerialNo` static field.

The `GetNextSerialNo` and `SetNextSerialNo` static methods can access the `nextSerialNo` static field, but it would be an error for them to directly access the `serialNo` instance field.

The following example shows the use of the `Entity` class.

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

Note that the `SetNextSerialNo` and `GetNextSerialNo` static methods are invoked on the class whereas the `GetSerialNo` instance method is invoked on instances of the class.

Virtual, override, and abstract methods

When an instance method declaration includes a `virtual` modifier, the method is said to be a ***virtual method***. When no `virtual` modifier is present, the method is said to be a ***non-virtual method***.

When a virtual method is invoked, the ***run-time type*** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the ***compile-time type*** of the instance is the determining factor.

A virtual method can be ***overridden*** in a derived class. When an instance method declaration includes an `override` modifier, the method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of that method.

An ***abstract*** method is a virtual method with no implementation. An abstract method is declared with the `abstract` modifier and is permitted only in a class that is also declared `abstract`. An abstract method must be overridden in every non-abstract derived class.

The following example declares an abstract class, `Expression`, which represents an expression tree node, and three derived classes, `Constant`, `VariableReference`, and `Operation`, which implement expression tree nodes for constants, variable references, and arithmetic operations. (This is similar to, but not to be confused with the expression tree types introduced in Expression tree types).

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

The previous four classes can be used to model arithmetic expressions. For example, using instances of these classes, the expression `x + 3` can be represented as follows.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

The `Evaluate` method of an `Expression` instance is invoked to evaluate the given expression and produce a `double` value. The method takes as an argument a `Hashtable` that contains variable names (as keys of the entries) and values (as values of the entries). The `Evaluate` method is a virtual abstract method, meaning that non-abstract derived classes must override it to provide an actual implementation.

A `Constant`'s implementation of `Evaluate` simply returns the stored constant. A `VariableReference`'s implementation looks up the variable name in the hashtable and returns the resulting value. An `Operation`'s implementation first evaluates the left and right operands (by recursively invoking their `Evaluate` methods) and then performs the given arithmetic operation.

The following program uses the `Expression` classes to evaluate the expression `x * (y + 2)` for different values of `x` and `y`.

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "16.5"
    }
}
```

Method overloading

Method ***overloading*** permits multiple methods in the same class to have the same name as long as they have unique signatures. When compiling an invocation of an overloaded method, the compiler uses ***overload resolution*** to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments or reports an error if no single best match can be found. The following example shows overload resolution in effect. The comment for each invocation in the `Main` method shows which method is actually invoked.

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

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

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

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                // Invokes F()
        F(1);               // Invokes F(int)
        F(1.0);             // Invokes F(double)
        F("abc");           // Invokes F(object)
        F((double)1);       // Invokes F(double)
        F((object)1);       // Invokes F(object)
        F<int>(1);          // Invokes F<T>(T)
        F(1, 1);            // Invokes F(double, double)
    }
}
```

As shown by the example, a particular method can always be selected by explicitly casting the arguments to the exact parameter types and/or explicitly supplying type arguments.

Other function members

Members that contain executable code are collectively known as the ***function members*** of a class. The preceding section describes methods, which are the primary kind of function members. This section describes the other kinds of function members supported by C#: constructors, properties, indexers, events, operators, and destructors.

The following code shows a generic class called `List<T>`, which implements a growable list of objects. The class contains several examples of the most common kinds of function members.

```
public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
```

```
        get { return count; }
    }
    public int Capacity {
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
}
```

Constructors

C# supports both instance and static constructors. An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. A ***static constructor*** is a member that implements the actions required to initialize a class itself when it is first loaded.

A constructor is declared like a method with no return type and the same name as the containing class. If a constructor declaration includes a `static` modifier, it declares a static constructor. Otherwise, it declares an instance constructor.

Instance constructors can be overloaded. For example, the `List<T>` class declares two instance constructors, one with no parameters and one that takes an `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `List<string>` instances using each of the constructors of the `List` class.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written.

A property is declared like a field, except that the declaration ends with a `get` accessor and/or a `set` accessor written between the delimiters `{` and `}` instead of ending in a semicolon. A property that has both a `get` accessor and a `set` accessor is a ***read-write property***, a property that has only a `get` accessor is a ***read-only property***, and a property that has only a `set` accessor is a ***write-only property***.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property.

A `set` accessor corresponds to a method with a single parameter named `value` and no return type. When a property is referenced as the target of an assignment or as the operand of `++` or `--`, the `set` accessor is invoked with an argument that provides the new value.

The `List<T>` class declares two properties, `Count` and `Capacity`, which are read-only and read-write, respectively. The following is an example of use of these properties.

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;            // Invokes get accessor
int j = names.Capacity;         // Invokes get accessor
```

Similar to fields and methods, C# supports both instance properties and static properties. Static properties are declared with the `static` modifier, and instance properties are declared without it.

The accessor(s) of a property can be virtual. When a property declaration includes a `virtual`, `abstract`, or `override` modifier, it applies to the accessor(s) of the property.

Indexers

An ***indexer*** is a member that enables objects to be indexed in the same way as an array. An indexer is declared like a property except that the name of the member is `this` followed by a parameter list written between the delimiters `[` and `]`. The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

The `List` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to

index `List` instances with `int` values. For example

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Indexers can be overloaded, meaning that a class can declare multiple indexers as long as the number or types of their parameters differ.

Events

An ***event*** is a member that enables a class or object to provide notifications. An event is declared like a field except that the declaration includes an `event` keyword and the type must be a delegate type.

Within a class that declares an event member, the event behaves just like a field of a delegate type (provided the event is not abstract and does not declare accessors). The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handles are present, the field is `null`.

The `List<T>` class declares a single event member called `Changed`, which indicates that a new item has been added to the list. The `Changed` event is raised by the `OnChanged` virtual method, which first checks whether the event is `null` (meaning that no handlers are present). The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events.

Clients react to events through ***event handlers***. Event handlers are attached using the `+=` operator and removed using the `-=` operator. The following example attaches an event handler to the `Changed` event of a `List<string>`.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);        // Outputs "3"
    }
}
```

For advanced scenarios where control of the underlying storage of an event is desired, an event declaration can explicitly provide `add` and `remove` accessors, which are somewhat similar to the `set` accessor of a property.

Operators

An ***operator*** is a member that defines the meaning of applying a particular expression operator to instances of a class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators. All operators must be declared as `public` and `static`.

The `List<T>` class declares two operators, `operator==` and `operator!=`, and thus gives new meaning to expressions that apply those operators to `List` instances. Specifically, the operators define equality of two

`List<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `List<int>` instances.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);        // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);        // Outputs "False"
    }
}
```

The first `Console.WriteLine` outputs `True` because the two lists contain the same number of objects with the same values in the same order. Had `List<T>` not defined `operator==`, the first `Console.WriteLine` would have output `False` because `a` and `b` reference different `List<int>` instances.

Destructors

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

The garbage collector is allowed wide latitude in deciding when to collect objects and run destructors. Specifically, the timing of destructor invocations is not deterministic, and destructors may be executed on any thread. For these and other reasons, classes should implement destructors only when no other solutions are feasible.

The `using` statement provides a better approach to object destruction.

Structs

Like classes, ***structs*** are data structures that can contain data members and function members, but unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs. For example, the following program creates and initializes an array of 100 points. With `Point` implemented as a class, 101 separate objects are instantiated—one for the array and one each for the 100 elements.

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

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

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

An alternative is to make `Point` a struct.

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

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

Now, only one object is instantiated—the one for the array—and the `Point` instances are stored in-line in the array.

Struct constructors are invoked with the `new` operator, but that does not imply that memory is being allocated. Instead of dynamically allocating an object and returning a reference to it, a struct constructor simply returns the struct value itself (typically in a temporary location on the stack), and this value is then copied as necessary.

With classes, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. For example, the output produced by the following code fragment depends on whether `Point` is a class or a struct.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

If `Point` is a class, the output is `20` because `a` and `b` reference the same object. If `Point` is a struct, the output is `10` because the assignment of `a` to `b` creates a copy of the value, and this copy is unaffected by the subsequent assignment to `a.x`.

The previous example highlights two of the limitations of structs. First, copying an entire struct is typically less efficient than copying an object reference, so assignment and value parameter passing can be more expensive with structs than with reference types. Second, except for `ref` and `out` parameters, it is not possible to create references to structs, which rules out their usage in a number of situations.

Arrays

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

Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at run-time using the `new` operator. The `new` operation specifies the ***length*** of the new array instance, which is then fixed for the lifetime of the instance. The indices of the elements of an array range from `0` to `Length - 1`. The `new` operator automatically initializes the elements of an array to their default value, which, for example, is zero for all numeric types and `null` for all reference types.

The following example creates an array of `int` elements, initializes the array, and prints out the contents of the array.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

This example creates and operates on a ***single-dimensional array***. C# also supports ***multi-dimensional arrays***. The number of dimensions of an array type, also known as the ***rank*** of the array type, is one plus the number of commas written between the square brackets of the array type. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10 × 5) elements, and the `a3` array contains 100 (10 × 5 × 2) elements.

The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a ***jagged array*** because the lengths of the element arrays do not all have to be the same. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The subsequent lines then initialize the three elements with references to individual array instances of varying lengths.

The `new` operator permits the initial values of the array elements to be specified using an ***array initializer***, which is a list of expressions written between the delimiters `{` and `}`. The following example allocates and initializes an `int[]` with three elements.

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

Note that the length of the array is inferred from the number of expressions between `{` and `}`. Local variable and field declarations can be shortened further such that the array type does not have to be restated.

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

Both of the previous examples are equivalent to the following:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Interfaces

An ***interface*** defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ ***multiple inheritance***. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

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

When a class or struct implements a particular interface, instances of that class or struct can be implicitly

converted to that interface type. For example

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

In cases where an instance is not statically known to implement a particular interface, dynamic type casts can be used. For example, the following statements use dynamic type casts to obtain an object's `IControl` and `IDataBound` interface implementations. Because the actual type of the object is `EditBox`, the casts succeed.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

In the previous `EditBox` class, the `Paint` method from the `IControl` interface and the `Bind` method from the `IDataBound` interface are implemented using `public` members. C# also supports ***explicit interface member implementations***, using which the class or struct can avoid making the members `public`. An explicit interface member implementation is written using the fully qualified interface member name. For example, the `EditBox` class could implement the `IControl.Paint` and `IDataBound.Bind` methods using explicit interface member implementations as follows.

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

Explicit interface members can only be accessed via the interface type. For example, the implementation of `IControl.Paint` provided by the previous `EditBox` class can only be invoked by first converting the `EditBox` reference to the `IControl` interface type.

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error, no such method
IControl control = editBox;
control.Paint();                        // Ok
```

Enums

An ***enum type*** is a distinct value type with a set of named constants. The following example declares and uses an enum type named `Color` with three constant values, `Red`, `Green`, and `Blue`.

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. An enum type that does not explicitly declare an underlying type has an underlying type of `int`. An enum type's storage format and range of possible values are determined by its underlying type. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type and is a distinct valid value of that enum type.

The following example declares an enum type named `Alignment` with an underlying type of `sbyte`.

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

As shown by the previous example, an enum member declaration can include a constant expression that specifies the value of the member. The constant value for each enum member must be in the range of the underlying type of the enum. When an enum member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the enum type) or the value of the textually preceding enum member plus one.

Enum values can be converted to integral values and vice versa using type casts. For example

```
int i = (int)Color.Blue;        // int i = 2;
Color c = (Color)2;             // Color c = Color.Blue;
```

The default value of any enum type is the integral value zero converted to the enum type. In cases where variables are automatically initialized to a default value, this is the value given to variables of enum types. In order for the default value of an enum type to be easily available, the literal `0` implicitly converts to any enum type. Thus, the following is permitted.

```
Color c = 0;
```

Delegates

A ***delegate type*** represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named `Function`.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

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

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

An instance of the `Function` delegate type can reference any method that takes a `double` argument and returns a `double` value. The `Apply` method applies a given `Function` to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, `Apply` is used to apply three different functions to a `double[]`.

A delegate can reference either a static method (such as `Square` or `Math.Sin` in the previous example) or an instance method (such as `m.Multiply` in the previous example). A delegate that references an instance method

also references a particular object, and when the instance method is invoked through the delegate, that object becomes `this` in the invocation.

Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. Thus, the multiplier example above can be written more easily without using a `Multiplier` class:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

Attributes

Types, members, and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a method is controlled using the `public`, `protected`, `internal`, and `private` modifiers. C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this additional declarative information by defining and using ***attributes***.

The following example declares a `HelpAttribute` attribute that can be placed on program entities to provide links to their associated documentation.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

All attribute classes derive from the `System.Attribute` base class provided by the .NET Framework. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` attribute can be used as follows.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

This example attaches a `HelpAttribute` to the `Widget` class and another `HelpAttribute` to the `Display` method in

the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The following example shows how attribute information for a given program entity can be retrieved at run-time using reflection.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

When a particular attribute is requested through reflection, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

Lexical structure

1/26/2018 • 33 minutes to read • Edit Online

Programs

A C# ***program*** consists of one or more ***source files***, known formally as ***compilation units*** (Compilation units). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding.

Conceptually speaking, a program is compiled using three steps:

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

Grammars

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

Grammar notation

The lexical and syntactic grammars are presented in Backus-Naur form using the notation of the ANTLR grammar tool.

Lexical grammar

The lexical grammar of C# is presented in Lexical analysis, Tokens, and Pre-processing directives. 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 (Tokens), white space (White space), comments (Comments), and pre-processing directives (Pre-processing directives).

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

Syntactic grammar

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

Every source file in a C# program must conform to the *compilation_unit* production of the syntactic grammar (Compilation units).

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?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* new_line
    | pp_directive
    ;

input_element
    : whitespace
    | comment
    | token
    ;
```

Five basic elements make up the lexical structure of a C# source file: Line terminators (Line terminators), white space (White space), comments (Comments), tokens (Tokens), and pre-processing directives (Pre-processing directives). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program (Syntactic grammar).

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.

In the case of interpolated string literals (Interpolated string literals) a single token is initially produced by lexical analysis, but is broken up into several input elements which are repeatedly subjected to lexical analysis until all interpolated string literals have been resolved. The resulting tokens then serve as input to the syntactic analysis.

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.

Line terminators

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

```
new_line
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Carriage return character (U+000D) followed by line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;
```

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

- If the last character of the source file is a Control-Z character (`U+001A`), this character is deleted.
- A carriage-return character (`U+000D`) is added to the end of the source file if that source file is 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`).

Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines.

```
comment
    : single_line_comment
    | delimited_comment
    ;

single_line_comment
    : '//' input_character*
    ;

input_character
    : '<Any Unicode character except a new_line_character>'
    ;

new_line_character
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

delimited_comment
    : '/*' delimited_comment_section* asterisk* '/'
    ;

delimited_comment_section
    : '/'
    | asterisk* not_slash_or_asterisk
    ;

asterisk
    : '*'
    ;

not_slash_or_asterisk
    : '<Any Unicode character except / or *>'
    ;
```

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

Comments are not processed within character and string literals.

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.

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.

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)>'
    ;
```

Tokens

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

```
token
    : identifier
    | keyword
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | interpolated_string_literal
    | operator_or_punctuator
    ;
```

Unicode character escape sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers (Identifiers), character literals (Character literals), and regular string literals (String literals). 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 code points in characters and string values, a Unicode character in the range U+10000 to U+10FFFF is not permitted in a character literal and is represented using a Unicode surrogate pair in a string literal. 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 "`\`". The Unicode value `\u005C` is the character "`\`".

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());
    }
}
```

Identifiers

The rules for identifiers given in this section correspond exactly to those recommended by the Unicode Standard Annex 31, 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_character*
    ;

identifier_start_character
    : letter_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>'
    ;
```

For information on the Unicode character classes mentioned above, see The Unicode Standard, Version 3.0, section 4.5.

Examples of valid identifiers include "`identifier1`", "`_identifier2`", and "`@if`".

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***. Use of the `@` prefix for identifiers that are not keywords is permitted, but strongly discouraged as a

matter of style.

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`".

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. For example, an implementation might provide extended keywords that begin with two underscores.

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
    : '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. Such identifiers are sometimes referred to as "contextual keywords". For example, within a property declaration, the "`get`" and "`set`"

identifiers have special meaning (Accessors). An identifier other than `get` or `set` is never permitted in these locations, so this use does not conflict with a use of these words as identifiers. In other cases, such as with the identifier "`var`" in implicitly typed local variable declarations (Local variable declarations), a contextual keyword can conflict with declared names. In such cases, the declared name takes precedence over the use of the identifier as a contextual keyword.

Literals

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

```
literal
    : boolean_literal
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | null_literal
    ;
```

Boolean literals

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

```
boolean_literal
    : 'true'
    | 'false'
    ;
```

The type of a *boolean_literal* is `bool`.

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_digit+ integer_type_suffix?
    ;

decimal_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;

integer_type_suffix
    : 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU' | 'lu'
    ;

hexadecimal_integer_literal
    : '0x' hex_digit+ integer_type_suffix?
    | '0X' hex_digit+ integer_type_suffix?
    ;

hex_digit
    : '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.

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

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

- When a *decimal_integer_literal* with the value 2147483648 (2^31) and no *integer_type_suffix* appears as the token immediately following a unary minus operator token (Unary minus operator), 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 (Unary minus operator), 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`.

Real literals

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

```
real_literal
    : decimal_digit+ '.' decimal_digit+ exponent_part? real_type_suffix?
    | '.' decimal_digit+ exponent_part? real_type_suffix?
    | decimal_digit+ exponent_part real_type_suffix?
    | decimal_digit+ real_type_suffix
    ;

exponent_part
    : 'e' sign? decimal_digit+
    | 'E' sign? decimal_digit+
    ;

sign
    : '+'
    | '-'
    ;

real_type_suffix
    : '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`. For example, the literals `1f`, `1.5f`, `1e10f`, and `123.456F` are all of type `float`.
- A real literal suffixed by `D` or `d` is of type `double`. For example, the literals `1d`, `1.5d`, `1e10d`, and `123.456D` are all of type `double`.
- A real literal suffixed by `M` or `m` is of type `decimal`. For example, the literals `1m`, `1.5m`, `1e10m`, and `123.456M`

are all of type `decimal`. 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 (The decimal type). 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). Hence, the literal `2.900m` will be parsed to form the decimal with sign `0`, coefficient `2900`, and scale `3`.

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

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

Note that in a real literal, decimal digits are always required after the decimal point. For example, `1.3F` is a real literal but `1.F` is not.

Character literals

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

Note: The ANTLR grammar notation makes the following confusing! In ANTLR, when you write `\'` it stands for a single quote `'`. And when you write `\\` it stands for a single backslash `\`. Therefore the first rule for a character literal means it starts with a single quote, then a character, then a single quote. And the eleven possible simple escape sequences are `\'`, `\"`, `\\`, `\0`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`.

```
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
    : '\\\'' | '\\"' | '\\\\' | '\\0' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v'
    ;

hexadecimal_escape_sequence
    : '\\x' hex_digit hex_digit? hex_digit? hex_digit?;
```

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

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

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

A Unicode character escape sequence (Unicode character escape sequences) 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`

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
`\"`	Double quote	`0x0022`
`\\`	Backslash	`0x005C`
`\0`	Null	`0x0000`
`\a`	Alert	`0x0007`
`\b`	Backspace	`0x0008`
`\f`	Form feed	`0x000C`
`\n`	New line	`0x000A`
`\r`	Carriage return	`0x000D`
`\t`	Horizontal tab	`0x0009`
`\v`	Vertical tab	`0x000B`

The type of a *character_literal* is `char`.

String literals

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

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

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

```
string_literal
    : regular_string_literal
    | verbatim_string_literal
    ;

regular_string_literal
    : '"' 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_character* '"'
    ;

verbatim_string_literal_character
    : single_verbatim_string_literal_character
    | quote_escape_sequence
    ;

single_verbatim_string_literal_character
    : '<any character except ">'
    ;

quote_escape_sequence
    : '""'
    ;
```

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

The example

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

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

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

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

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

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

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

The type of a *string_literal* is `string`.

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (String equality operators) appear in the same program, these string literals refer to the same string instance. 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.

Interpolated string literals

Interpolated string literals are similar to string literals, but contain holes delimited by `{` and `}`, wherein expressions can occur. At runtime, the expressions are evaluated with the purpose of having their textual forms substituted into the string at the place where the hole occurs. The syntax and semantics of string interpolation are described in section (Interpolated strings).

Like string literals, interpolated string literals can be either regular or verbatim. Interpolated regular string literals are delimited by `$"` and `"`, and interpolated verbatim string literals are delimited by `$@"` and `"`.

Like other literals, lexical analysis of an interpolated string literal initially results in a single token, as per the grammar below. However, before syntactic analysis, the single token of an interpolated string literal is broken into several tokens for the parts of the string enclosing the holes, and the input elements occurring in the holes are lexically analysed again. This may in turn produce more interpolated string literals to be processed, but, if lexically correct, will eventually lead to a sequence of tokens for syntactic analysis to process.

```
interpolated_string_literal
    : '$' interpolated_regular_string_literal
    | '$' interpolated_verbatim_string_literal
    ;

interpolated_regular_string_literal
    : interpolated_regular_string_whole
    | interpolated_regular_string_start  interpolated_regular_string_literal_body
interpolated_regular_string_end
    ;

interpolated_regular_string_literal_body
    : regular_balanced_text
    | interpolated_regular_string_literal_body interpolated_regular_string_mid regular_balanced_text
    ;

interpolated_regular_string_whole
    : '"' interpolated_regular_string_character* '"'
    ;

interpolated_regular_string_start
    : '"' interpolated_regular_string_character* '{'
    ;

interpolated_regular_string_mid
```

```
    : interpolation_format? '}' interpolated_regular_string_characters_after_brace? '{'
    ;

interpolated_regular_string_end
    : interpolation_format? '}' interpolated_regular_string_characters_after_brace? '""'
    ;

interpolated_regular_string_characters_after_brace
    : interpolated_regular_string_character_no_brace
    | interpolated_regular_string_characters_after_brace interpolated_regular_string_character
    ;

interpolated_regular_string_character
    : single_interpolated_regular_string_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_regular_string_character_no_brace
    : '<Any interpolated_regular_string_character except close_brace_escape_sequence and any
hexadecimal_escape_sequence or unicode_escape_sequence designating } (U+007D)>'
    ;

single_interpolated_regular_string_character
    : '<Any character except \" (U+0022), \\ (U+005C), { (U+007B), } (U+007D), and new_line_character>'
    ;

open_brace_escape_sequence
    : '{{'
    ;

    close_brace_escape_sequence
    : '}}'
    ;

regular_balanced_text
    : regular_balanced_text_part+
    ;

regular_balanced_text_part
    : single_regular_balanced_text_character
    | delimited_comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' regular_balanced_text ')'
    | '[' regular_balanced_text ']'
    | '{' regular_balanced_text '}'
    ;

single_regular_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B), } (U+007D) and new_line_character>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;

interpolation_format
    : interpolation_format_character+
    ;

interpolation_format_character
    : '<Any character except \" (U+0022), : (U+003A), { (U+007B) and } (U+007D)>'
    ;

interpolated_verbatim_string_literal
    : interpolated_verbatim_string_whole
```

```
    | interpolated_verbatim_string_start interpolated_verbatim_string_literal_body
interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_literal_body
    : verbatim_balanced_text
    | interpolated_verbatim_string_literal_body interpolated_verbatim_string_mid verbatim_balanced_text
    ;

interpolated_verbatim_string_whole
    : '@"' interpolated_verbatim_string_character* '"'
    ;

interpolated_verbatim_string_start
    : '@"' interpolated_verbatim_string_character* '{'
    ;

interpolated_verbatim_string_mid
    : interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '{'
    ;

interpolated_verbatim_string_end
    : interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '"'
    ;

interpolated_verbatim_string_characters_after_brace
    : interpolated_verbatim_string_character_no_brace
    | interpolated_verbatim_string_characters_after_brace interpolated_verbatim_string_character
    ;

interpolated_verbatim_string_character
    : single_interpolated_verbatim_string_character
    | quote_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_verbatim_string_character_no_brace
    : '<Any interpolated_verbatim_string_character except close_brace_escape_sequence>'
    ;

single_interpolated_verbatim_string_character
    : '<Any character except \" (U+0022), { (U+007B) and } (U+007D)>'
    ;

verbatim_balanced_text
    : verbatim_balanced_text_part+
    ;

verbatim_balanced_text_part
    : single_verbatim_balanced_text_character
    | comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' verbatim_balanced_text ')'
    | '[' verbatim_balanced_text ']'
    | '{' verbatim_balanced_text '}'
    ;

single_verbatim_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B) and } (U+007D)>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;
```

An *interpolated_string_literal* token is reinterpreted as multiple tokens and other input elements as follows, in

order of occurrence in the *interpolated_string_literal*:

- Occurrences of the following are reinterpreted as separate individual tokens: the leading `$` sign, *interpolated_regular_string_whole*, *interpolated_regular_string_start*, *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_whole*, *interpolated_verbatim_string_start*, *interpolated_verbatim_string_mid* and *interpolated_verbatim_string_end*.
- Occurrences of *regular_balanced_text* and *verbatim_balanced_text* between these are reprocessed as an *input_section* (Lexical analysis) and are reinterpreted as the resulting sequence of input elements. These may in turn include interpolated string literal tokens to be reinterpreted.

Syntactic analysis will recombine the tokens into an *interpolated_string_expression* (Interpolated strings).

Examples TODO

The null literal

```
null_literal
    : 'null'
    ;
```

The *null_literal* can be implicitly converted to a reference type or nullable type.

Operators and punctuators

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

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

right_shift
    : '>>'
    ;

right_shift_assignment
    : '>>='
    ;
```

The vertical bar in the *right_shift* and *right_shift_assignment* productions are used to indicate that, unlike other productions in the syntactic grammar, no characters of any kind (not even whitespace) are allowed between the tokens. These productions are treated specially in order to enable the correct handling of *type_parameter_lists* (Type parameters).

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. 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.

```
pp_directive
    : pp_declaration
    | pp_conditional
    | pp_line
    | pp_diagnostic
    | pp_region
    | pp_pragma
    ;
```

The following pre-processing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation symbols (Declaration directives).
- `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code (Conditional compilation directives).
- `#line`, which is used to control line numbers emitted for errors and warnings (Line directives).
- `#error` and `#warning`, which are used to issue errors and warnings, respectively (Diagnostic directives).
- `#region` and `#endregion`, which are used to explicitly mark sections of source code (Region directives).
- `#pragma`, which is used to specify optional contextual information to the compiler (Pragma directives).

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

A source line containing a `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, `#line`, or `#endregion` directive 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.

Conditional compilation symbols

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

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`. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the value `false`.

The 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.

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? pp_or_expression whitespace?
    ;

pp_or_expression
    : pp_and_expression
    | pp_or_expression whitespace? '||' whitespace? pp_and_expression
    ;

pp_and_expression
    : pp_equality_expression
    | pp_and_expression whitespace? '&&' whitespace? pp_equality_expression
    ;

pp_equality_expression
    : pp_unary_expression
    | pp_equality_expression whitespace? '==' whitespace? pp_unary_expression
    | pp_equality_expression whitespace? '!=' whitespace? pp_unary_expression
    ;

pp_unary_expression
    : pp_primary_expression
    | '!' whitespace? pp_unary_expression
    ;

pp_primary_expression
    : 'true'
    | 'false'
    | conditional_symbol
    | '(' whitespace? pp_expression whitespace? ')'
    ;
```

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 (Constant expressions), except that the only user-defined entities that can be referenced are conditional compilation symbols.

Declaration directives

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

```
pp_declaration
    : whitespace? '#' whitespace? 'define' whitespace conditional_symbol pp_new_line
    | whitespace? '#' whitespace? 'undef' whitespace conditional_symbol pp_new_line
    ;

pp_new_line
    : whitespace? single_line_comment? 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* (Tokens) 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.

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.

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
}
```

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

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

A `#undef` may "undefine" a conditional compilation symbol that is not defined. 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
```

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_section* pp_else_section? pp_endif
    ;

pp_if_section
    : whitespace? '#' whitespace? 'if' whitespace pp_expression pp_new_line conditional_section?
    ;

pp_elif_section
    : whitespace? '#' whitespace? 'elif' whitespace pp_expression pp_new_line conditional_section?
    ;

pp_else_section:
    | whitespace? '#' whitespace? 'else' pp_new_line conditional_section?
    ;

pp_endif
    : whitespace? '#' whitespace? 'endif' pp_new_line
    ;

conditional_section
    : input_section
    | skipped_section
    ;

skipped_section
    : skipped_section_part+
    ;

skipped_section_part
    : skipped_characters? new_line
    | pp_directive
    ;

skipped_characters
    : whitespace? not_number_sign input_character*
    ;

not_number_sign
    : '<Any input_character except #>'
    ;
```

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.

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.

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.

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? '#' whitespace? 'error' pp_message
    | whitespace? '#' whitespace? 'warning' pp_message
    ;

pp_message
    : new_line
    | whitespace input_character* new_line
    ;
```

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 ("A build can't be both debug and retail") if the conditional symbols `Debug` and `Retail` are both defined. Note that a *pp_message* can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the word `can't` .

Region directives

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

```
pp_region
    : pp_start_region conditional_section? pp_end_region
    ;

pp_start_region
    : whitespace? '#' whitespace? 'region' pp_message
    ;

pp_end_region
    : whitespace? '#' whitespace? '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
```

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, and that are used by caller info attributes (Caller info attributes).

Line directives are most commonly used in meta-programming tools that generate C# source code from some other text input.

```
pp_line
    : whitespace? '#' whitespace? 'line' whitespace line_indicator pp_new_line
    ;

line_indicator
    : decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;

file_name
    : '"' file_name_character+ '"'
    ;

file_name_character
    : '<Any input_character except ">'
    ;
```

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.

A `#line hidden` directive has no effect on the file and line numbers reported in error messages, but does affect source level debugging. When debugging, all lines between a `#line hidden` directive and the subsequent `#line` directive (that is not `#line hidden`) have no line number information. When stepping through code in the debugger, these lines will be skipped entirely.

Note that a *file_name* differs from a regular string literal in that escape characters are not processed; the "`\`" character simply designates an ordinary backslash character within a *file_name*.

Pragma directives

The `#pragma` preprocessing directive is used to specify optional contextual information to the compiler. The information supplied in a `#pragma` directive will never change program semantics.

```
pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

C# provides `#pragma` directives to control compiler warnings. Future versions of the language may include additional `#pragma` directives. To ensure interoperability with other C# compilers, the Microsoft C# compiler does not issue compilation errors for unknown `#pragma` directives; such directives do however generate warnings.

Pragma warning

The `#pragma warning` directive is used to disable or restore all or a particular set of warning messages during compilation of the subsequent program text.

```
pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

A `#pragma warning` directive that omits the warning list affects all warnings. A `#pragma warning` directive the includes a warning list affects only those warnings that are specified in the list.

A `#pragma warning disable` directive disables all or the given set of warnings.

A `#pragma warning restore` directive restores all or the given set of warnings to the state that was in effect at the beginning of the compilation unit. Note that if a particular warning was disabled externally, a `#pragma warning restore` (whether for all or the specific warning) will not re-enable that warning.

The following example shows use of `#pragma warning` to temporarily disable the warning reported when obsoleted members are referenced, using the warning number from the Microsoft C# compiler.

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
    Foo();
#pragma warning restore 612
    }
}
```

Basic concepts

1/26/2018 • 48 minutes to read • Edit Online

Application Startup

An assembly that has an ***entry point*** is called an ***application***. When an application is run, a new ***application domain*** is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same 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.

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

```
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 (Application termination).

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

Since 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 may contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or their only parameter is other than type `string[]`.

An application can be made up of multiple classes 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, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility (Declared accessibility) of a method is determined by the access modifiers (Access modifiers) 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.

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

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

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).

Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (Namespaces), which can contain type declarations and nested namespace declarations. Type declarations (Type declarations) are used to define classes (Classes), structs (Structs), interfaces (Interfaces), enums (Enums), and delegates (Delegates). 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 (Constants), fields (Fields), methods (Methods), properties (Properties), events (Events), indexers (Indexers), operators (Operators), instance constructors (Instance constructors), static constructors (Static constructors), destructors (Destructors), and nested types (Nested types).

A declaration defines a name in the ***declaration space*** to which the declaration belongs. Except for overloaded members (Signatures and overloading), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. 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, *interface_member_declaration*s, or *type_parameter*s. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded 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 (Signatures and overloading). 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 delegate declaration creates a new declaration space. Names are introduced into this declaration space through formal parameters (*fixed_parameter*s and *parameter_array*s) and *type_parameter*s.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum_member_declarations*.
- Each method declaration, indexer declaration, operator declaration, instance constructor declaration and anonymous function creates a new declaration space called a ***local variable declaration space***. Names are introduced into this declaration space through formal parameters (*fixed_parameter*s and *parameter_array*s) and *type_parameter*s. The body of the function member or anonymous function, if any, is considered to be nested within the local variable declaration space. It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.
- Each *block* or *switch_block* , as well as a *for*, *foreach* and *using* statement, creates a local variable declaration space for local variables and local constants . Names are introduced into this declaration space through *local_variable_declaration*s and *local_constant_declaration*s. Note that blocks that occur as or within the body of a function member or anonymous function are nested within the local variable declaration space declared by those functions for their parameters. Thus it is an error to have e.g. a method with a local variable and a parameter of the same name.
- Each *block* or *switch_block* creates a separate declaration space for labels. Names are introduced into this declaration space through *labeled_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 (Scopes).
- Declaration order for enum member declarations (Enum members) is significant when *constant_expression* values are omitted.

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.

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();
    }
}
```

Members

Namespaces and types have ***members***. 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.

Members of a type are either declared in the type declaration or ***inherited*** from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, 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 (Declared accessibility) or because it is hidden by a declaration in the type itself (Hiding through inheritance).

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.

Struct members

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

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

- 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.

Enumeration members

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

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.

Interface members

The members of an interface are the members declared in the interface and in all base interfaces of the interface. The members in class `object` are not, strictly speaking, members of any interface (Interface members). However, the members in class `object` are available via member lookup in any interface type (Member lookup).

Array members

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

Delegate members

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

Member access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (Declared accessibility) 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 (Accessibility domains) of the member.

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 (meaning protected or 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.

Accessibility domains

The ***accessibility domain*** of a member consists of the (possibly disjoint) sections of program text in which access

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

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

The accessibility domain of a top-level unbound type `T` (Bound and unbound types) 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`.

From these definitions it follows that the accessibility domain of a top-level unbound type is always at least the program text of the program in which that type is declared.

The accessibility domain for a constructed type `T<A1, ..., An>` is the intersection of the accessibility domain of the unbound generic type `T` and the accessibility domains of the type arguments `A1, ..., An`.

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`.

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.

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 (Protected access for instance members).
- 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 (Protected access for instance members).
- 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.

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

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

As described in Members, 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. 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.

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

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

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

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a *constructor_initializer* (Constructor initializers).

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`.

In the example

```
class C<T>
{
    protected T x;
}

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

the three assignments to `x` are permitted because they all take place through instances of class types constructed from the generic type.

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.

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`.

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.

Signatures and overloading

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

- The signature of a method consists of the name of the method, the number of type parameters and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method. The signature of a method specifically does not include the return type, the `params` modifier that may be specified for the right-most parameter, nor the optional type parameter constraints.
- 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, nor does it include the `params` modifier that may be specified for the right-most parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

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

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

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

For the purposes of signatures, the types `object` and `dynamic` are considered the same. Members declared in a single type can therefore not differ in signature solely by `object` and `dynamic`.

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)      error

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

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 (this does not, however, remove the restriction imposed by Declarations 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). 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* (Namespace members) 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 name defined by an *extern_alias_directive* extends over the *using_directives*, *global_attributes* and *namespace_member_declarations* of its immediately containing compilation unit or namespace body. An *extern_alias_directive* does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a *using_directive* (Using directives) 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, type or member names available within a particular *compilation_unit* or *namespace_body*, but does not contribute any new members to the underlying declaration space. In other words, a *using_directive* is not transitive but rather affects only the *compilation_unit* or *namespace_body* in which it occurs.
- The scope of a type parameter declared by a *type_parameter_list* on a *class_declaration* (Class declarations) is

the *class_base*, *type_parameter_constraints_clause*s, and *class_body* of that *class_declaration*.

- The scope of a type parameter declared by a *type_parameter_list* on a *struct_declaration* (Struct declarations) is the *struct_interfaces*, *type_parameter_constraints_clause*s, and *struct_body* of that *struct_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on an *interface_declaration* (Interface declarations) is the *interface_base*, *type_parameter_constraints_clause*s, and *interface_body* of that *interface_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on a *delegate_declaration* (Delegate declarations) is the *return_type*, *formal_parameter_list*, and *type_parameter_constraints_clause*s of that *delegate_declaration*.
- The scope of a member declared by a *class_member_declaration* (Class body) 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 (Accessibility domains) of the member.
- The scope of a member declared by a *struct_member_declaration* (Struct members) is the *struct_body* in which the declaration occurs.
- The scope of a member declared by an *enum_member_declaration* (Enum members) is the *enum_body* in which the declaration occurs.
- The scope of a parameter declared in a *method_declaration* (Methods) is the *method_body* of that *method_declaration*.
- The scope of a parameter declared in an *indexer_declaration* (Indexers) is the *accessor_declarations* of that *indexer_declaration*.
- The scope of a parameter declared in an *operator_declaration* (Operators) is the *block* of that *operator_declaration*.
- The scope of a parameter declared in a *constructor_declaration* (Instance constructors) is the *constructor_initializer* and *block* of that *constructor_declaration*.
- The scope of a parameter declared in a *lambda_expression* (Anonymous function expressions) is the *anonymous_function_body* of that *lambda_expression*
- The scope of a parameter declared in an *anonymous_method_expression* (Anonymous function expressions) is the *block* of that *anonymous_method_expression*.
- The scope of a label declared in a *labeled_statement* (Labeled statements) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local_variable_declaration* (Local variable declarations) is the block in which the declaration occurs.
- The scope of a local variable declared in a *switch_block* of a `switch` statement (The switch statement) is the *switch_block*.
- The scope of a local variable declared in a *for_initializer* of a `for` statement (The for statement) 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* (Local constant declarations) is the block in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant_declarator*.
- The scope of a variable declared as part of a *foreach_statement*, *using_statement*, *lock_statement* or *query_expression* is determined by the expansion of the given construct.

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

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. 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*.

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);                   // writes "A"
    }
}
```

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

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.

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.

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.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. 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.

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 (Operators) 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. 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.

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

```
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.

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

```
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`.

Namespace and type names

Several contexts in a C# program require a *namespace_name* or a *type_name* to be specified.

```
namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;
```

A *namespace_name* is a *namespace_or_type_name* that refers to a namespace. Following resolution as described below, the *namespace_or_type_name* of a *namespace_name* must refer to a namespace, or otherwise a compile-time error occurs. No type arguments (Type arguments) can be present in a *namespace_name* (only types can have type arguments).

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

If the *namespace_or_type_name* is a qualified-alias-member its meaning is as described in Namespace alias qualifiers. Otherwise, a *namespace_or_type_name* has one of four forms:

- `I`
- `I<A1, ..., Ak>`
- `N.I`
- `N.I<A1, ..., Ak>`

where `I` is a single identifier, `N` is a *namespace_or_type_name* and `<A1, ..., Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `k` to be zero.

The meaning of a *namespace_or_type_name* is determined as follows:

- If the *namespace_or_type_name* is of the form `I` or of the form `I<A1, ..., Ak>`:
  - If `K` is zero and the *namespace_or_type_name* appears within a generic method declaration (Methods) and if that declaration includes a type parameter (Type parameters) with name `I`, then the *namespace_or_type_name* refers to that type parameter.
  - Otherwise, if the *namespace_or_type_name* appears within a type declaration, then for each instance type `T` (The instance type), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
    - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *namespace_or_type_name* refers to that type parameter.
    - Otherwise, if the *namespace_or_type_name* appears within the body of the type declaration, and `T` or any of its base types contain a nested accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace_or_type_name*.
  - If the previous steps were unsuccessful then, for each namespace `N`, starting with the namespace in which the *namespace_or_type_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
    - If `K` is zero and `I` is the name of a namespace in `N`, then:
      - If the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
      - Otherwise, the *namespace_or_type_name* refers to the namespace named `I` in `N`.
    - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
      - If `K` is zero and the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
      - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
    - Otherwise, if the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N`:
      - If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type,

then the *namespace_or_type_name* refers to that namespace or type.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_alias_directive*s of the namespace declaration contain exactly one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_alias_directive*s of the namespace declaration contain more than one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* is ambiguous and an error occurs.
  - Otherwise, the *namespace_or_type_name* is undefined and a compile-time error occurs.
- Otherwise, the *namespace_or_type_name* is of the form `N.I` or of the form `N.I<A1, ..., Ak>`. `N` is first resolved as a *namespace_or_type_name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<A1, ..., Ak>` is resolved as follows:
  - If `K` is zero and `N` refers to a namespace and `N` contains a nested namespace with name `I`, then the *namespace_or_type_name* refers to that nested namespace.
  - Otherwise, if `N` refers to a namespace and `N` contains an accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
  - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `N` or any of its base classes contain a nested accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that if the meaning of `N.I` is being determined as part of resolving the base class specification of `N` then the direct base class of `N` is considered to be object (Base classes).
  - Otherwise, `N.I` is an invalid *namespace_or_type_name*, and a compile-time error occurs.

A *namespace_or_type_name* is permitted to reference a static class (Static classes) only if

- The *namespace_or_type_name* is the `T` in a *namespace_or_type_name* of the form `T.I`, or
- The *namespace_or_type_name* is the `T` in a *typeof_expression* (Argument lists1) of the form `typeof(T)`.

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.

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
}
```

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. The C# compiler and the garbage collector 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, the garbage collector may (but is not required to) treat the object as no longer in use.
3. Once the object is eligible for destruction, at some unspecified later time the destructor (Destructors) (if any) for the object is run. Under normal circumstances the destructor for the object is run once only, though implementation-specific APIs may allow this behavior to be overridden.
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.

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 `A` before the destructor of `B`, 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 `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not, because of the reference `Test.RefA`.

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.

An alternative to using destructors is to let a class implement the `System.IDisposable` interface. This allows the client of the object to determine when to release the resources of the object, typically by accessing the object as a resource in a `using` statement (The using statement).

Execution order

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

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

Types

1/13/2018 • 34 minutes to read • Edit Online

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

```
type
    : value_type
    | reference_type
    | type_parameter
    | type_unsafe
    ;
```

The final category of types, pointers, is available only in unsafe code. This is discussed further in Pointer 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 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 (Boxing and unboxing).

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
    | nullable_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

nullable_type
    : non_nullable_value_type '?'
    ;

non_nullable_value_type
    : type
    ;

enum_type
    : type_name
    ;
```

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

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

The System.ValueType type

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

Note that `System.ValueType` is not itself a *value_type*. Rather, it is a *class_type* from which all *value_type*s are automatically derived.

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`, converted to the type `E`.
- For a *struct_type*, the default value is the value produced by setting all value type fields to their default value and all reference type fields to `null`.
- For a *nullable_type* the default value is an instance for which the `HasValue` property is false and the `Value` property is undefined. The default value is also known as the ***null value*** of the nullable type.

Like any other instance constructor, the default constructor of a value type is invoked using the `new` operator. For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. 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 (Constructors).

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. The declaration of struct types is described in Struct declarations.

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`

RESERVED WORD	ALIASED TYPE
`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* (Literals). 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* (Constant expressions). 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 (Constants). 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 (Evaluation of user-defined conversions).

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. Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other.

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_literals* or as *integer_literals* in combination with a cast to type `char`. For example, `(char)10` is the same as `'\x000A'`.

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

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 (Division operator).
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.
- The ***Not-a-Number*** value, often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form `s * m * 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 <= e <= 104`, and for `double`,

`0 < m < 2^53` and `1075 <= e <= 970`. Denormalized floating-point numbers are considered valid non-zero values.

The `float` type can represent values ranging from approximately `1.5 * 10^-45` to `3.4 * 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.

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 * c * 10^-e`, where the sign `s` is 0 or 1, the coefficient `c` is given by `0 <= *c* < 2^96`, and the scale `e` is such that `0 <= e <= 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 `1.0m`, the value is exact to 28 or 29 digits. Contrary to the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

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

The result of an operation on values of type `decimal` is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). A zero result always has a sign of 0 and a scale of 0.

If a decimal arithmetic operation produces a value less than or equal to `5 * 10^-29` in absolute value, 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.

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.

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`.

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

Nullable types

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

A ***non-nullable value type*** conversely is any value type other than `System.Nullable<T>` and its shorthand `T?` (for any `T`), plus any type parameter that is constrained to be a non-nullable value type (that is, any type parameter with a `struct` constraint). The `System.Nullable<T>` type specifies the value type constraint for `T` (Type parameter constraints), which means that the underlying type of a nullable type can be any non-nullable value type. The underlying type of a nullable type cannot be a nullable type or a reference type. For example, `int??` and `string?` are invalid types.

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

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

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

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

In addition to the default constructor, every nullable type `T?` has a public constructor that takes a single argument of type `T`. Given a value `x` of type `T`, a constructor invocation of the form

```
new T?(x)
```

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

Implicit conversions are available from the `null` literal to `T?` (Null literal conversions) and from `T` to `T?` (Implicit nullable conversions).

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'
    | 'dynamic'
    | 'string'
    ;

interface_type
    : type_name
    ;

array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;

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.

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* (Object creation expressions).

Class types are described in Classes.

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

CLASS TYPE	DESCRIPTION
`System.Object`	The ultimate base class of all other types. See The object type.
`System.String`	The string type of the C# language. See The string type.
`System.ValueType`	The base class of all value types. See The System.ValueType type.
`System.Enum`	The base class of all enum types. See Enums.
`System.Array`	The base class of all array types. See Arrays.
`System.Delegate`	The base class of all delegate types. See Delegates.
`System.Exception`	The base class of all exception types. See Exceptions.

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`.

The dynamic type

The `dynamic` type, like `object`, can reference any object. When operators are applied to expressions of type `dynamic`, their resolution is deferred until the program is run. Thus, if the operator cannot legally be applied to the referenced object, no error is given during compilation. Instead an exception will be thrown when resolution of the operator fails at run-time.

Its purpose is to allow dynamic binding, which is described in detail in Dynamic binding.

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

- Operations on expressions of type `dynamic` can be dynamically bound (Dynamic binding).
- Type inference (Type inference) will prefer `dynamic` over `object` if both are candidates.

Because of this equivalence, the following holds:

- There is an implicit identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing `dynamic` with `object`
- Implicit and explicit conversions to and from `object` also apply to and from `dynamic`.
- Method signatures that are the same when replacing `dynamic` with `object` are considered the same signature
- The type `dynamic` is indistinguishable from `object` at run-time.

- An expression of the type `dynamic` is referred to as a ***dynamic expression***.

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 (String literals).

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

Interface types

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

Interface types are described in Interfaces.

Array types

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

Array types are described in Arrays.

Delegate types

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

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

Delegate types are described in Delegates.

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.

Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a *reference_type*. The following boxing conversions exist:

- From any *value_type* to the type `object`.
- From any *value_type* to the type `System.ValueType`.
- From any *non_nullable_value_type* to any *interface_type* implemented by the *value_type*.
- From any *nullable_type* to any *interface_type* implemented by the underlying type of the *nullable_type*.
- From any *enum_type* to the type `System.Enum`.
- From any *nullable_type* with an underlying *enum_type* to the type `System.Enum`.
- Note that an implicit conversion from a type parameter will be executed as a boxing conversion if at run-time it ends up converting from a value type to a reference type (Implicit conversions involving type parameters).

Boxing a value of a *non_nullable_value_type* consists of allocating an object instance and copying the *non_nullable_value_type* value into that instance.

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

The actual process of boxing a value of a *non_nullable_value_type* is best explained by imagining the existence of a generic ***boxing class***, which behaves as if it were declared as follows:

```
sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

Boxing of a value `v` of type `T` now consists of executing the expression `new Box<T>(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 Box<int>(i);
```

A boxing class like `Box<T>` above doesn'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`. For example,

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

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

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

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

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

the following statements

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

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance.

Unboxing conversions

An unboxing conversion permits a *reference_type* to be explicitly converted to a *value_type*. The following unboxing conversions exist:

- From the type `object` to any *value_type*.
- From the type `System.ValueType` to any *value_type*.
- From any *interface_type* to any *non_nullable_value_type* that implements the *interface_type*.
- From any *interface_type* to any *nullable_type* whose underlying type implements the *interface_type*.
- From the type `System.Enum` to any *enum_type*.
- From the type `System.Enum` to any *nullable_type* with an underlying *enum_type*.
- Note that an explicit conversion to a type parameter will be executed as an unboxing conversion if at run-time it ends up converting from a reference type to a value type (Explicit dynamic conversions).

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

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

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 `((Box<T>)box).value`. Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

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

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

Constructed types

A generic type declaration, by itself, denotes an ***unbound generic type*** that is used as a "blueprint" to form many different types, by way of applying ***type arguments***. The type arguments are written within angle brackets (`<` and `>`) immediately following the name of the generic type. A type that includes at least one type argument is called a

constructed type. A constructed type can be used in most places in the language in which a type name can appear. An unbound generic type can only be used within a *typeof_expression* (The typeof operator).

Constructed types can also be used in expressions as simple names (Simple names) or when accessing a member (Member access).

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

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;

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

A *type_name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup (Nested types in generic classes):

```
class Outer<T>
{
    public class Inner {...}

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

In unsafe code, a constructed type cannot be used as an *unmanaged_type* (Pointer types).

Type arguments

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

```
type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;
```

In unsafe code (Unsafe code), a *type_argument* may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter (Type parameter constraints).

Open and closed types

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

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

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

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

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

Bound and unbound types

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

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

Satisfying constraints

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

- If the constraint is a class type, an interface type, or a type parameter, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type `A` is convertible to type `C` by one of the following:
  - An identity conversion (Identity conversion)
  - An implicit reference conversion (Implicit reference conversions)
  - A boxing conversion (Boxing conversions), provided that type A is a non-nullable value type.
  - An implicit reference, boxing or type parameter conversion from a type parameter `A` to `C`.
- If the constraint is the reference type constraint (`class`), the type `A` must satisfy one of the following:
  - `A` is an interface type, class type, delegate type or array type. Note that `System.ValueType` and `System.Enum` are reference types that satisfy this constraint.
  - `A` is a type parameter that is known to be a reference type (Type parameter constraints).
- If the constraint is the value type constraint (`struct`), the type `A` must satisfy one of the following:
  - `A` is a struct type or enum type, but not a nullable type. Note that `System.ValueType` and `System.Enum` are reference types that do not satisfy this constraint.
  - `A` is a type parameter having the value type constraint (Type parameter constraints).
- If the constraint is the constructor constraint `new()`, the type `A` must not be `abstract` and must have a public parameterless constructor. This is satisfied if one of the following is true:
  - `A` is a value type, since all value types have a public default constructor (Default constructors).

- `A` is a type parameter having the constructor constraint (Type parameter constraints).
- `A` is a type parameter having the value type constraint (Type parameter constraints).
- `A` is a class that is not `abstract` and contains an explicitly declared `public` constructor with no parameters.
- `A` is not `abstract` and has a default constructor (Default constructors).

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

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

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

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

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

Type parameters

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

```
type_parameter
    : identifier
    ;
```

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

- A type parameter cannot be used directly to declare a base class (Base class) or interface (Variant type parameter lists).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in Member lookup.
- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in Implicit conversions involving type parameters and Explicit dynamic conversions.
- The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type (Implicit conversions involving type parameters). However, a `default` expression (Default value expressions) can be used instead. In addition, a value with a type given by a type parameter can be compared with `null` using `==` and `!=` (Reference type equality operators) unless the type parameter has the value type constraint.
- A `new` expression (Object creation expressions) can only be used with a type parameter if the type parameter is constrained by a *constructor_constraint* or the value type constraint (Type parameter constraints).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access (Member access) or type name (Namespace and type names) to identify a static member or a nested type.
- In unsafe code, a type parameter cannot be used as an *unmanaged_type* (Pointer types).

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed constructed type (Open and closed types). The

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

Expression tree types

Expression trees permit lambda expressions to be represented as data structures instead of executable code. Expression trees are values of ***expression tree types*** of the form `System.Linq.Expressions.Expression<D>`, where `D` is any delegate type. For the remainder of this specification we will refer to these types using the shorthand `Expression<D>`.

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

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

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

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

```
Func<int,int> del = x => x + 1;                  // Code

Expression<Func<int,int>> exp = x => x + 1;      // Data
```

Following these assignments, the delegate `del` references a method that returns `x + 1`, and the expression tree `exp` references a data structure that describes the expression `x => x + 1`.

The exact definition of the generic type `Expression<D>` as well as the precise rules for constructing an expression tree when a lambda expression is converted to an expression tree type, are both outside the scope of this specification.

Two things are important to make explicit:

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

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

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

  ```
  int i1 = del(1);

  int i2 = del2(1);
  ```

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

Variables

1/26/2018 • 32 minutes to read • Edit Online

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*** (Definite assignment) before its value can be obtained.

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

Variable 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.

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.

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 (Static constructors) 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 (Default values) of the variable's type.

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

Instance variables

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

Instance variables in classes

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

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

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

Instance variables in structs

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

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

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 (Default values) of the type of the array elements.

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

Value parameters

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

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

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

Reference parameters

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

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

The following definite assignment rules apply to reference parameters. Note the different rules for output parameters described in Output parameters.

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

Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a reference parameter of the struct type (This access).

Output parameters

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

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

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters described in Reference parameters.

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

Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of the struct type (This access).

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*; or by a *foreach_statement* or a *specific_catch_clause* for a *try_statement*.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends at least from entry into the *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* with which it is associated, until execution of that *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* ends in any way. (Entering an enclosed *block* or calling a method suspends, but does not end, execution of the current *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause*.) If the local variable is captured by an anonymous function (Captured outer variables), its lifetime extends at least until the delegate or expression tree created from the anonymous function, along with any other objects that come to reference the captured variable, are eligible for garbage collection.

If the parent *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* is entered recursively, a new instance of the local variable is created each time, and its *local_variable_initializer*, if any, is evaluated each time.

A local variable introduced by a *local_variable_declaration* is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable introduced by a *local_variable_declaration* is considered initially unassigned. A *local_variable_declaration* may include a *local_variable_initializer*, in which case the variable is considered definitely assigned only after the initializing expression (Declaration statements).

Within the scope of a local variable introduced by a *local_variable_declaration*, it is a compile-time error to refer to that local variable in a textual position that precedes its *local_variable_declarator*. If the local variable declaration is implicit (Local variable declarations), it is also an error to refer to the variable within its *local_variable_declarator*.

A local variable introduced by a *foreach_statement* or a *specific_catch_clause* is considered definitely assigned in its entire scope.

The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is only used for a small portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable (Automatic memory management).

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 (Default constructors).
- For a variable of a *reference_type*, the default value is `null`.

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

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 a particular static flow analysis (Precise rules for determining definite assignment), that the variable has been automatically initialized or has been the target of at least one assignment. Informally stated, the rules of definite assignment are:

- An initially assigned variable (Initially assigned variables) is always considered definitely assigned.
- An initially unassigned variable (Initially unassigned variables) 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 (Simple assignment) in which the variable is the left operand.
  - An invocation expression (Invocation expressions) or object creation expression (Object creation expressions) that passes the variable as an output parameter.
  - For a local variable, a local variable declaration (Local variable declarations) that includes a variable initializer.

The formal specification underlying the above informal rules is described in Initially assigned variables, Initially unassigned variables, and Precise rules for determining definite assignment.

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

- An instance variable is considered definitely assigned if its containing *struct_type* variable is considered definitely assigned.
- A *struct_type* variable is considered definitely assigned if each of its instance variables is considered definitely assigned.

Definite assignment is a requirement in the following contexts:

- A variable must be definitely assigned at each location where its value is obtained. This ensures that undefined values never occur. The occurrence of a variable in an expression is considered to obtain the value of the variable, except when
  - the variable is the left operand of a simple assignment,
  - the variable is passed as an output parameter, or
  - the variable is a *struct_type* variable and occurs as the left operand of a member access.
- A variable must be definitely assigned at each location where it is passed as a reference parameter. This ensures that the function member being invoked can consider the reference parameter initially assigned.
- All output parameters of a function member must be definitely assigned at each location where the function member returns (through a `return` statement or through execution reaching the end of the function member

body). This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable.

- The `this` variable of a *struct_type* instance constructor must be definitely assigned at each location where that instance constructor returns.

Initially assigned 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.

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.

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 (End points and reachability) of each statement
- On each arc which transfers control to another statement or to the end point of a statement
- At the beginning of each expression
- At the end of each expression

The definite assignment state of *v* can be either:

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

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

General rules for statements

- *v* is not definitely assigned at the beginning of a function member body.
- *v* is definitely assigned at the beginning of any unreachable statement.
- The definite assignment state of *v* at the beginning of any other statement is determined by checking the

definite assignment state of *v* on all control flow transfers that target the beginning of that statement. If (and only if) *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (End points and reachability).

- 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 (End points and reachability).

Block statements, checked, and unchecked statements

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

Expression statements

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

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

Declaration statements

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

If statements

For an `if` statement *stmt* of the form:

```
if ( expr ) then_stmt else else_stmt
```

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

Switch statements

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

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

While statements

For a `while` statement *stmt* of the form:

```
while ( expr ) while_body
```

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

Do statements

For a `do` statement *stmt* of the form:

```
do do_body while ( expr ) ;
```

- *v* has the same definite assignment state on the control flow transfer from the beginning of *stmt* to *do_body* as at the beginning of *stmt*.
- *v* has the same definite assignment state at the beginning of *expr* as at the end point of *do_body*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.

For statements

Definite assignment checking for a `for` statement of the form:

```
for ( 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.

Break, continue, and goto statements

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

Throw statements

For a statement *stmt* of the form

```
throw expr ;
```

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

Return statements

For a statement *stmt* of the form

```
return expr ;
```

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

For a statement stmt of the form:

```
return ;
```

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

Try-catch statements

For a statement *stmt* of the form:

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
```

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

Try-finally statements

For a `try` statement *stmt* of the form:

```
try try_block finally finally_block
```

- The definite assignment state of *v* at the beginning of *try_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *finally_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) at least one of

the following is true:

- *v* is definitely assigned at the end-point of *try_block*
- *v* is definitely assigned at the end-point of *finally_block*

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

Try-catch-finally statements

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

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
finally *finally_block*
```

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

```
try {
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
}
finally finally_block
```

The following example demonstrates how the different blocks of a `try` statement (The try statement) 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

}
}
```

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*.

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*.

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*.

Yield statements

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

```
yield return expr ;
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end of *stmt* is the same as the state of *v* at the end of *expr*.
- A `yield break` statement has no effect on the definite assignment state.

General rules for simple expressions

The following rule applies to these kinds of expressions: literals (Literals), simple names (Simple names), member access expressions (Member access), non-indexed base access expressions (Base access), `typeof` expressions (The typeof operator), default value expressions (Default value expressions) and `nameof` expressions (Nameof expressions).

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

General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions (Parenthesized expressions), element access expressions (Element access), base access expressions with indexing (Base access), increment and decrement expressions (Postfix increment and decrement operators, Prefix increment and decrement operators), cast expressions (Cast expressions), unary `+`, `-`, `~`, `*` expressions, binary `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions (Arithmetic operators, Shift operators, Relational and type-

testing operators, Logical operators), compound assignment expressions (Compound assignment), `checked` and `unchecked` expressions (The checked and unchecked operators), plus array and delegate creation expressions (The new operator).

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 *e1, e2, ..., eN*, evaluated in that order:

- The definite assignment state of *v* at the beginning of *e1* is the same as the definite assignment state at the beginning of *expr*.
- The definite assignment state of *v* at the beginning of *ei* (*i* greater than one) is the same as the definite assignment state at the end of the previous sub-expression.
- The definite assignment state of *v* at the end of *expr* is the same as the definite assignment state at the end of *eN*

Invocation expressions and object creation expressions

For an invocation expression *expr* of the form:

```
primary_expression ( arg1 , arg2 , ... , argN )
```

or an object creation expression of the form:

```
new type ( arg1 , arg2 , ... , argN )
```

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

Simple assignment expressions

For an expression *expr* of the form `w = expr_rhs`:

- The definite assignment state of *v* before *expr_rhs* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
  - If *w* is the same variable as *v*, then the definite assignment state of *v* after *expr* is definitely assigned.
  - Otherwise, if the assignment occurs within the instance constructor of a struct type, if *w* is a property access designating an automatically implemented property *P* on the instance being constructed and *v* is the hidden backing field of *P*, 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*.

&& (conditional AND) expressions

For an expression *expr* of the form `expr_first && expr_second`:

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

In the example

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

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

|| (conditional OR) expressions

For an expression *expr* of the form `expr_first || expr_second`:

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

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

In the example

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

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

! (logical negation) 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".

?? (null coalescing) expressions

For an expression *expr* of the form `expr_first ?? expr_second`:

- The definite assignment state of *v* before *expr_first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr_second* is the same as the definite assignment state of *v* after *expr_first*.
- The definite assignment statement of *v* after *expr* is determined by:
  - If *expr_first* is a constant expression (Constant expressions) with value null, then the the state of *v* after *expr* is the same as the state of *v* after *expr_second*.

- Otherwise, the state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_first*.

?: (conditional) expressions

For an expression *expr* of the form `expr_cond ? expr_true : expr_false`:

- The definite assignment state of *v* before *expr_cond* is the same as the state of *v* before *expr*.
- The definite assignment state of *v* before *expr_true* is definitely assigned if and only if one of the following holds:
  - *expr_cond* is a constant expression with the value `false`
  - the state of *v* after *expr_cond* is definitely assigned or "definitely assigned after true expression".
- The definite assignment state of *v* before *expr_false* is definitely assigned if and only if one of the following holds:
  - *expr_cond* is a constant expression with the value `true`
- the state of *v* after *expr_cond* is definitely assigned or "definitely assigned after false expression".
- The definite assignment state of *v* after *expr* is determined by:
  - If *expr_cond* is a constant expression (Constant expressions) with value `true` then the state of *v* after *expr* is the same as the state of *v* after *expr_true*.
  - Otherwise, if *expr_cond* is a constant expression (Constant expressions) with value `false` then the state of *v* after *expr* is the same as the state of *v* after *expr_false*.
  - Otherwise, if the state of *v* after *expr_true* is definitely assigned and the state of *v* after *expr_false* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

Anonymous functions

For a *lambda_expression* or *anonymous_method_expression expr* with a body (either *block* or *expression*) *body*:

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

The example

```
delegate bool Filter(int i);

void F() {
    int max;

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

    max = 5;
    DoWork(f);
}
```

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

```
delegate void D();

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

    d();

    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

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

Variable references

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

```
variable_reference
    : expression
    ;
```

In C and C++, a *variable_reference* is known as an *lvalue*.

Atomicity of variable references

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

Conversions

1/13/2018 • 47 minutes to read • Edit Online

A ***conversion*** enables an expression to be treated as being of a particular type. A conversion may cause an expression of a given type to be treated as having a different type, or it may cause an expression without a type to get a type. Conversions can be ***implicit*** or ***explicit***, and this determines whether an explicit cast is required. 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
```

Some conversions are defined by the language. Programs may also define their own conversions (User-defined conversions).

Implicit conversions

The following conversions are classified as implicit conversions:

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

Implicit conversions can occur in a variety of situations, including function member invocations (Compile-time checking of dynamic overload resolution), cast expressions (Cast expressions), and assignments (Assignment operators).

The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. Properly designed user-defined implicit conversions should exhibit these characteristics as well.

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

However, dynamic conversions (Implicit dynamic conversions and Explicit dynamic conversions) apply only to expressions of type `dynamic` (The dynamic type).

Identity conversion

An identity conversion converts from any type to the same type. This conversion exists such that an entity that already has a required type can be said to be convertible to that type.

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

Implicit numeric conversions

The implicit numeric conversions are:

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

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

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

Implicit enumeration conversions

An implicit enumeration conversion permits the *decimal_integer_literal* `0` to be converted to any *enum_type* and to any *nullable_type* whose underlying type is an *enum_type*. In the latter case the conversion is evaluated by converting to the underlying *enum_type* and wrapping the result (Nullable types).

Implicit interpolated string conversions

An implicit interpolated string conversion permits an *interpolated_string_expression* (Interpolated strings) to be converted to `System.IFormattable` or `System.FormattableString` (which implements `System.IFormattable`).

When this conversion is applied a string value is not composed from the interpolated string. Instead an instance of `System.FormattableString` is created, as further described in Interpolated strings.

Implicit nullable conversions

Predefined implicit conversions that operate on non-nullable value types can also be used with nullable forms of those types. For each of the predefined implicit identity and numeric conversions that convert from a non-nullable value type `S` to a non-nullable value type `T`, the following implicit nullable conversions exist:

- An implicit conversion from `S?` to `T?`.
- An implicit conversion from `S` to `T?`.

Evaluation of an implicit nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
  - If the source value is null (`HasValue` property is false), the result is the null value of type `T?`.
  - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping (Nullable types) from `T` to `T?`.
- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from

`S` to `T` followed by a wrapping from `T` to `T?`.

Null literal conversions

An implicit conversion exists from the `null` literal to any nullable type. This conversion produces the null value (Nullable types) of the given nullable type.

Implicit reference conversions

The implicit reference conversions are:

- From any *reference_type* to `object` and `dynamic`.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is derived from `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` implements `T`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is derived from `T`.
- From an *array_type* `S` with an element type `SE` to an *array_type* `T` with an element type `TE`, 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 `SE` and `TE` are *reference_type*s.
  - An implicit reference conversion exists from `SE` to `TE`.
- From any *array_type* to `System.Array` and the interfaces it implements.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its base interfaces, provided that there is an implicit identity or reference conversion from `S` to `T`.
- From any *delegate_type* to `System.Delegate` and the interfaces it implements.
- From the null literal to any *reference_type*.
- From any *reference_type* to a *reference_type* `T` if it has an implicit identity or reference conversion to a *reference_type* `T0` and `T0` has an identity conversion to `T`.
- From any *reference_type* to an interface or delegate type `T` if it has an implicit identity or reference conversion to an interface or delegate type `T0` and `T0` is variance-convertible (Variance conversion) to `T`.
- Implicit conversions involving type parameters that are known to be reference types. See Implicit conversions involving type parameters for more details on implicit conversions involving type parameters.

The implicit reference conversions are those conversions between *reference_type*s 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. 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.

Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a reference type. A boxing conversion exists from any *non_nullable_value_type* to `object` and `dynamic`, to `System.ValueType` and to any *interface_type* implemented by the *non_nullable_value_type*. Furthermore an *enum_type* can be converted to the type `System.Enum`.

A boxing conversion exists from a *nullable_type* to a reference type, if and only if a boxing conversion exists from the underlying *non_nullable_value_type* to the reference type.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface type `I0` and `I0` has an identity conversion to `I`.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface or delegate type `I0` and `I0` is variance-convertible (Variance conversion) to `I`.

Boxing a value of a *non_nullable_value_type* consists of allocating an object instance and copying the *value_type*

value into that instance. A struct can be boxed to the type `System.ValueType`, since that is a base class for all structs (Inheritance).

Boxing a value of a *nullable_type* proceeds as follows:

- If the source value is null (`HasValue` property is false), the result is a null reference of the target type.
- Otherwise, the result is a reference to a boxed `T` produced by unwrapping and boxing the source value.

Boxing conversions are described further in Boxing conversions.

Implicit dynamic conversions

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

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

The following example illustrates implicit dynamic conversions:

```
object o  = "object"
dynamic d = "dynamic";

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

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

Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A *constant_expression* (Constant expressions) 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.

Implicit conversions involving type parameters

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

- From `T` to its effective base class `C`, from `T` to any base class of `C`, and from `T` to any interface implemented by `C`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From `T` to an interface type `I` in `T`'s effective interface set and from `T` to any base interface of `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From `T` to a type parameter `U`, provided `T` depends on `U` (Type parameter constraints). At run-time, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an

implicit reference conversion or identity conversion.

- From the null literal to `T`, provided `T` is known to be a reference type.
- From `T` to a reference type `I` if it has an implicit conversion to a reference type `S0` and `S0` has an identity conversion to `S`. At run-time the conversion is executed the same way as the conversion to `S0`.
- From `T` to an interface type `I` if it has an implicit conversion to an interface or delegate type `I0` and `I0` is variance-convertible to `I` (Variance conversion). At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

If `T` is known to be a reference type (Type parameter constraints), the conversions above are all classified as implicit reference conversions (Implicit reference conversions). If `T` is not known to be a reference type, the conversions above are classified as boxing conversions (Boxing conversions).

User-defined implicit conversions

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

Anonymous function conversions and method group conversions

Anonymous functions and method groups do not have types in and of themselves, but may be implicitly converted to delegate types or expression tree types. Anonymous function conversions are described in more detail in Anonymous function conversions and method group conversions in Method group conversions.

Explicit conversions

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit nullable conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- Explicit dynamic conversions
- User-defined explicit conversions.

Explicit conversions can occur in cast expressions (Cast expressions).

The set of explicit conversions includes all implicit conversions. This means that redundant cast expressions are allowed.

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.

Explicit numeric conversions

The explicit numeric conversions are the conversions from a *numeric_type* to another *numeric_type* for which an implicit numeric conversion (Implicit numeric conversions) 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 (Cast expressions).

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 (The checked and unchecked operators) 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 (The checked and unchecked operators) in which the conversion takes place:
  - In a `checked` context, the conversion proceeds as follows:
    - If the value of the operand is NaN or infinite, a `System.OverflowException` is thrown.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.
    - Otherwise, a `System.OverflowException` is thrown.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
    - If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.

    - Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes 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 28th decimal place if required (The decimal type). 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.

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 (Explicit numeric conversions) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion (Implicit numeric conversions) from `byte` to `int`.

Explicit nullable conversions

Explicit nullable conversions permit predefined explicit conversions that operate on non-nullable value types to also be used with nullable forms of those types. For each of the predefined explicit conversions that convert from a non-nullable value type `S` to a non-nullable value type `T` (Identity conversion, Implicit numeric conversions, Implicit enumeration conversions, Explicit numeric conversions, and Explicit enumeration conversions), the following nullable conversions exist:

- An explicit conversion from `S?` to `T?`.
- An explicit conversion from `S` to `T?`.
- An explicit conversion from `S?` to `T`.

Evaluation of a nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
    - If the source value is null (`HasValue` property is false), the result is the null value of type `T?`.
    - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S?` to `T`, the conversion is evaluated as an unwrapping from `S?` to `S` followed by the underlying conversion from `S` to `T`.

Note that an attempt to unwrap a nullable value will throw an exception if the value is `null`.

Explicit reference conversions

The explicit reference conversions are:

- From `object` and `dynamic` to any other *reference_type*.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is a base class of `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface_type* `S` to any *class_type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is not derived from `T`.
- From an *array_type* `S` with an element type `SE` to an *array_type* `T` with an element type `TE`, 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 `SE` and `TE` are *reference_type*s.
  - An explicit reference conversion exists from `SE` to `TE`.
- From `System.Array` and the interfaces it implements to any *array_type*.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its base interfaces, provided that there is an explicit reference conversion from `S` to `T`.
- From `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]`, provided that there is an explicit identity or reference conversion from `S` to `T`.
- From `System.Delegate` and the interfaces it implements to any *delegate_type*.
- From a reference type to a reference type `T` if it has an explicit reference conversion to a reference type `T0` and `T0` has an identity conversion `T`.
- From a reference type to an interface or delegate type `T` if it has an explicit reference conversion to an interface or delegate type `T0` and either `T0` is variance-convertible to `T` or `T` is variance-convertible to `T0` (Variance conversion).
- From `D<S1...Sn>` to `D<T1...Tn>` where `D<X1...Xn>` is a generic delegate type, `D<S1...Sn>` is not compatible with or identical to `D<T1...Tn>`, and for each type parameter `Xi` of `D` the following holds:
  - If `Xi` is invariant, then `Si` is identical to `Ti`.
  - If `Xi` is covariant, then there is an implicit or explicit identity or reference conversion from `Si` to `Ti`.
  - If `Xi` is contravariant, then `Si` and `Ti` are either identical or both reference types.
- Explicit conversions involving type parameters that are known to be reference types. For more details on explicit conversions involving type parameters, see Explicit conversions involving type parameters.

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 (Implicit reference conversions) or boxing conversion (Boxing conversions). 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. 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.

Unboxing conversions

An unboxing conversion permits a reference type to be explicitly converted to a *value_type*. An unboxing conversion exists from the types `object`, `dynamic` and `System.ValueType` to any *non_nullable_value_type*, and from any *interface_type* to any *non_nullable_value_type* that implements the *interface_type*. Furthermore type `System.Enum` can be unboxed to any *enum_type*.

An unboxing conversion exists from a reference type to a *nullable_type* if an unboxing conversion exists from the

reference type to the underlying *non_nullable_value_type* of the *nullable_type*.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface type `I0` and `I0` has an identity conversion to `I`.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface or delegate type `I0` and either `I0` is variance-convertible to `I` or `I` is variance-convertible to `I0` (Variance conversion).

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 a null reference to a *nullable_type* produces the null value of the *nullable_type*. A struct can be unboxed from the type `System.ValueType`, since that is a base class for all structs (Inheritance).

Unboxing conversions are described further in Unboxing conversions.

Explicit dynamic conversions

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

If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

Assume the following class is defined:

```
class C
{
    int i;

    public C(int i) { this.i = i; }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

The following example illustrates explicit dynamic conversions:

```
object o  = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

The best conversion of `o` to `C` is found at compile-time to be an explicit reference conversion. This fails at run-time, because `"1"` is not in fact a `C`. The conversion of `d` to `C` however, as an explicit dynamic conversion, is suspended to run-time, where a user defined conversion from the run-time type of `d` -- `string` -- to `C` is found, and succeeds.

Explicit conversions involving type parameters

The following explicit conversions exist for a given type parameter `T`:

- From the effective base class `C` of `T` to `T` and from any base class of `C` to `T`. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.

- From any interface type to `T`. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From `T` to any *interface_type* `I` provided there is not already an implicit conversion from `T` to `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion followed by an explicit reference conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From a type parameter `U` to `T`, provided `T` depends on `U` (Type parameter constraints). At run-time, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.

If `T` is known to be a reference type, the conversions above are all classified as explicit reference conversions (Explicit reference conversions). If `T` is not known to be a reference type, the conversions above are classified as unboxing conversions (Unboxing conversions).

The above rules do not permit a direct explicit conversion from an unconstrained type parameter to a non-interface type, which might be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear. For example, consider the following declaration:

```
class X<T>
{
    public static long F(T t) {
        return (long)t;               // Error
    }
}
```

If the direct explicit conversion of `t` to `int` were permitted, one might easily expect that `X<int>.F(7)` would return `7L`. However, it would not, because the standard numeric conversions are only considered when the types are known to be numeric at binding-time. In order to make the semantics clear, the above example must instead be written:

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;       // Ok, but will only work when T is long
    }
}
```

This code will now compile but executing `X<int>.F(7)` would then throw an exception at run-time, since a boxed `int` cannot be converted directly to a `long`.

User-defined explicit conversions

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

Standard conversions

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (Identity conversion)
- Implicit numeric conversions (Implicit numeric conversions)

- Implicit nullable conversions (Implicit nullable conversions)
- Implicit reference conversions (Implicit reference conversions)
- Boxing conversions (Boxing conversions)
- Implicit constant expression conversions (Implicit dynamic conversions)
- Implicit conversions involving type parameters (Implicit conversions involving type parameters)

The standard implicit conversions specifically exclude user-defined implicit conversions.

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. 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`.

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 (Conversion operators) in class and struct types.

Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let `S0` and `T0` refer to their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S0` and `T0` are different types.
- Either `S0` or `T0` is the class or struct type in which the operator declaration takes place.
- Neither `S0` nor `T0` is an *interface_type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

The restrictions that apply to user-defined conversions are discussed further in Conversion operators.

Lifted conversion operators

Given a user-defined conversion operator that converts from a non-nullable value type `S` to a non-nullable value type `T`, a ***lifted conversion operator*** exists that converts from `S?` to `T?`. This lifted conversion operator performs an unwrapping from `S?` to `S` followed by the user-defined conversion from `S` to `T` followed by a wrapping from `T` to `T?`, except that a null valued `S?` converts directly to a null valued `T?`.

A lifted conversion operator has the same implicit or explicit classification as its underlying user-defined conversion operator. The term "user-defined conversion" applies to the use of both user-defined and lifted conversion operators.

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). For the purposes of this step, if either the source or target type is a *nullable_type*, their underlying

type is used instead.

- From that set of types, determining which user-defined and lifted conversion operators are applicable. For a conversion operator to be applicable, it must be possible to perform a standard conversion (Standard conversions) 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. User-defined conversion operators are preferred over lifted conversion operators. The exact rules for establishing the most specific user-defined conversion operator are defined in the following 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 or lifted conversion operator.
- Next, invoking the user-defined or lifted conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined or lifted conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined or lifted conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following sections. The definitions make use of the following terms:

- If a standard implicit conversion (Standard implicit conversions) 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.

Processing of user-defined implicit conversions

A user-defined implicit conversion from type `S` to type `T` is processed as follows:

- Determine the types `S0` and `T0`. If `S` or `T` are nullable types, `S0` and `T0` are their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively.
- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S0` (if `S0` is a class or struct), the base classes of `S0` (if `S0` is a class), and `T0` (if `T0` is a class or struct).
- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted 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, `SX`, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then `SX` is `S`.
  - Otherwise, `SX` is the most encompassed type in the combined set of source types of the operators in `U`.

If exactly one most encompassed type cannot be found, then the conversion is ambiguous and a compile-time error occurs.

- Find the most specific target type, `TX`, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then `TX` is `T`.
  - Otherwise, `TX` is the most encompassing type in the combined set of target types of the operators in `U`. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
  - If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
  - Otherwise, if `U` contains exactly one lifted conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
  - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
  - If `S` is not `SX`, then a standard implicit conversion from `S` to `SX` is performed.
  - The most specific conversion operator is invoked to convert from `SX` to `TX`.
  - If `TX` is not `T`, then a standard implicit conversion from `TX` to `T` is performed.

Processing of user-defined explicit conversions

A user-defined explicit conversion from type `S` to type `T` is processed as follows:

- Determine the types `S0` and `T0`. If `S` or `T` are nullable types, `S0` and `T0` are their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively.
- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S0` (if `S0` is a class or struct), the base classes of `S0` (if `S0` is a class), `T0` (if `T0` is a class or struct), and the base classes of `T0` (if `T0` is a class).
- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted 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, `SX`, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then `SX` is `S`.
  - Otherwise, if any of the operators in `U` convert from types that encompass `S`, then `SX` 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, `SX` is the most encompassing type in the combined set of source types of the operators in `U`. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, `TX`, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then `TX` is `T`.
  - Otherwise, if any of the operators in `U` convert to types that are encompassed by `T`, then `TX` is the most encompassing type in the combined set of target types of those operators. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, `TX` is the most encompassed type in the combined set of target types of the operators in `U`. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
  - If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.

    - Otherwise, if `U` contains exactly one lifted conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
    - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
    - If `S` is not `SX`, then a standard explicit conversion from `S` to `SX` is performed.
    - The most specific user-defined conversion operator is invoked to convert from `SX` to `TX`.
    - If `TX` is not `T`, then a standard explicit conversion from `TX` to `T` is performed.

Anonymous function conversions

An *anonymous_method_expression* or *lambda_expression* is classified as an anonymous function (Anonymous function expressions). The expression does not have a type but can be implicitly converted to a compatible delegate type or expression tree type. Specifically, an anonymous function `F` is compatible with a delegate type `D` provided:

- If `F` contains an *anonymous_function_signature*, then `D` and `F` have the same number of parameters.
- If `F` does not contain an *anonymous_function_signature*, then `D` may have zero or more parameters of any type, as long as no parameter of `D` has the `out` parameter modifier.
- If `F` has an explicitly typed parameter list, each parameter in `D` has the same type and modifiers as the corresponding parameter in `F`.
- If `F` has an implicitly typed parameter list, `D` has no `ref` or `out` parameters.
- If the body of `F` is an expression, and either `D` has a `void` return type or `F` is async and `D` has the return type `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt Expressions) that would be permitted as a *statement_expression* (Expression statements).
- If the body of `F` is a statement block, and either `D` has a `void` return type or `F` is async and `D` has the return type `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt Blocks) in which no `return` statement specifies an expression.
- If the body of `F` is an expression, and *either* `F` is non-async and `D` has a non-void return type `T`, *or* `F` is async and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt Expressions) that is implicitly convertible to `T`.
- If the body of `F` is a statement block, and *either* `F` is non-async and `D` has a non-void return type `T`, *or* `F` is async and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt Blocks) with a non-reachable end point in which each `return` statement specifies an expression that is implicitly convertible to `T`.

For the purpose of brevity, this section uses the short form for the task types `Task` and `Task<T>` (Async functions).

A lambda expression `F` is compatible with an expression tree type `Expression<D>` if `F` is compatible with the delegate type `D`. Note that this does not apply to anonymous methods, only lambda expressions.

Certain lambda expressions cannot be converted to expression tree types: Even though the conversion *exists*, it fails at compile-time. This is the case if the lambda expression:

- Has a *block* body
- Contains simple or compound assignment operators
- Contains a dynamically bound expression
- Is async

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

```
delegate R Func<A,R>(A arg);
```

In the assignments

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

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

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

Func<int, Task<int>> f4 = async x => x + 1;    // Ok
```

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

The first assignment successfully converts the anonymous function to the delegate type `Func<int,int>` because, when `x` is given type `int`, `x+1` is a valid expression that is implicitly convertible to type `int`.

Likewise, the second assignment successfully converts the anonymous function to the delegate type `Func<int,double>` because the result of `x+1` (of type `int`) is implicitly convertible to type `double`.

However, the third assignment is a compile-time error because, when `x` is given type `double`, the result of `x+1` (of type `double`) is not implicitly convertible to type `int`.

The fourth assignment successfully converts the anonymous async function to the delegate type `Func<int, Task<int>>` because the result of `x+1` (of type `int`) is implicitly convertible to the result type `int` of the task type `Task<int>`.

Anonymous functions may influence overload resolution, and participate in type inference. See Function members for further details.

Evaluation of anonymous function conversions to delegate types

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

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

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

```
delegate double Function(double x);

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

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

Since the two anonymous function delegates have the same (empty) set of captured outer variables, and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous function expressions.

Evaluation of anonymous function conversions to expression tree types

Conversion of an anonymous function to an expression tree type produces an expression tree (Expression tree types). More precisely, evaluation of the anonymous function conversion leads to the construction of an object structure that represents the structure of the anonymous function itself. The precise structure of the expression tree, as well as the exact process for creating it, are implementation defined.

Implementation example

This section describes a possible implementation of anonymous function conversions in terms of other C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation, nor is it the only one possible. It only briefly mentions conversions to expression trees, as their exact semantics are outside the scope of this specification.

The remainder of this section gives several examples of code that contains anonymous functions with different characteristics. For each example, a corresponding translation to code that uses only other C# constructs is provided. In the examples, the identifier `D` is assumed by represent the following delegate type:

```
public delegate void D();
```

The simplest form of an anonymous function is one that captures no outer variables:

```
class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}
```

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

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }

    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

In the following example, the anonymous function references instance members of `this`:

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

This can be translated to a compiler generated instance method containing the code of the anonymous function:

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```

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

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

The lifetime of the local variable must now be extended to at least the lifetime of the anonymous function delegate. This can be achieved by "hoisting" the local variable into a field of a compiler generated class. Instantiation of the local variable (Instantiation of local variables) then corresponds to creating an instance of the compiler generated class, and accessing the local variable corresponds to accessing a field in the instance of the compiler generated class. Furthermore, the anonymous function becomes an instance method of the compiler generated class:

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

Finally, the following anonymous function captures `this` as well as two local variables with different lifetimes:

```
class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```

Here, a compiler generated class is created for each statement block in which locals are captured such that the locals in the different blocks can have independent lifetimes. An instance of `__Locals2`, the compiler generated class for the inner statement block, contains the local variable `z` and a field that references an instance of `__Locals1`. An instance of `__Locals1`, the compiler generated class for the outer statement block, contains the local variable `y` and a field that references `this` of the enclosing function member. With these data structures it is possible to reach all captured outer variables through an instance of `__Local2`, and the code of the anonymous function can thus be implemented as an instance method of that class.

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }

    class __Locals1
    {
        public Test __this;
        public int y;
    }

    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;

        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}
```

The same technique applied here to capture local variables can also be used when converting anonymous functions to expression trees: References to the compiler generated objects can be stored in the expression tree, and access to the local variables can be represented as field accesses on these objects. The advantage of this approach is that it allows the "lifted" local variables to be shared between delegates and expression trees.

Method group conversions

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

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

- A single method `M` is selected corresponding to a method invocation (Method invocations) of the form `E(A)`, with the following modifications:
  - The argument list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref` or `out`) of the corresponding parameter in the *formal_parameter_list* of `D`.
  - The candidate methods considered are only those methods that are applicable in their normal form (Applicable function member), not those applicable only in their expanded form.
- If the algorithm of Method invocations produces an error, then a compile-time error occurs. Otherwise the algorithm produces a single best method `M` having the same number of parameters as `D` and the conversion is considered to exist.
- The selected method `M` must be compatible (Delegate compatibility) with the delegate type `D`, or otherwise, a

compile-time error occurs.

- If the selected method `M` is an instance method, the instance expression associated with `E` determines the target object of the delegate.
- If the selected method M is an extension method which is denoted by means of a member access on an instance expression, that instance expression determines the target object of the delegate.
- The result of the conversion is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.
- Note that this process can lead to the creation of a delegate to an extension method, if the algorithm of Method invocations fails to find an instance method but succeeds in processing the invocation of `E(A)` as an extension method invocation (Extension method invocations). A delegate thus created captures the extension method as well as its first argument.

The following example demonstrates method group conversions:

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

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

delegate string D5(int i);

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

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

    }
}
```

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

The assignment to `d2` shows how it is possible to create a delegate to a method that has less derived (contra-variant) parameter types and a more derived (covariant) return type.

The assignment to `d3` shows how no conversion exists if the method is not applicable.

The assignment to `d4` shows how the method must be applicable in its normal form.

The assignment to `d5` shows how parameter and return types of the delegate and method are allowed to differ only for reference types.

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a method group conversion. Thus, the example

```
object obj = new EventHandler(myDialog.OkClick);
```

could instead be written

```
object obj = (EventHandler)myDialog.OkClick;
```

Method groups may influence overload resolution, and participate in type inference. See Function members for further details.

The run-time evaluation of a method group conversion proceeds as follows:

- If the method selected at compile-time is an instance method, or it is an extension method which is accessed as an instance method, the target object of the delegate is determined from the instance expression associated with `E`:
  - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the instance expression is of a *reference_type*, the value computed by the instance expression becomes the target object. If the selected method is an instance method and the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - If the instance expression is of a *value_type*, a boxing operation (Boxing conversions) is performed to convert the value to an object, and this object becomes the target object.
- Otherwise the selected method is part of a static method call, and the target object of the delegate is `null`.
- A 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.

Expressions

1/13/2018 • 227 minutes to read • Edit Online

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

Expression classifications

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left hand side of a *member_access* (Member access). 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* (Member access), or as an operand for the `as` operator (The as operator), the `is` operator (The is operator), or the `typeof` operator (The typeof operator). 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 (Member lookup). A method group may have an associated instance expression and an associated type argument list. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (This access). A method group is permitted in an *invocation_expression* (Invocation expressions) , a *delegate_creation_expression* (Delegate creation expressions) and as the left hand side of an is operator, and can be implicitly converted to a compatible delegate type (Method group conversions). In any other context, an expression classified as a method group causes a compile-time error.
- A null literal. An expression with this classification can be implicitly converted to a reference type or nullable type.
- An anonymous function. An expression with this classification can be implicitly converted to a compatible delegate type or expression tree type.
- A property access. Every property access has an associated type, namely the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the `get` or `set` block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (This access).
- 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 (Event assignment). 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` (This access), 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* (Expression statements).

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 (Simple assignment). Otherwise, the *get accessor* is invoked to obtain the current value (Values of expressions).

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 (Definite assignment) 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 (Compile-time checking of dynamic overload resolution) 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 (Compile-time checking of dynamic overload resolution) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

Static and Dynamic Binding

The process of determining the meaning of an operation based on the type or value of constituent expressions (arguments, operands, receivers) is often referred to as ***binding***. For instance the meaning of a method call is determined based on the type of the receiver and arguments. The meaning of an operator is determined based on the type of its operands.

In C# the meaning of an operation is usually determined at compile-time, based on the compile-time type of its constituent expressions. Likewise, if an expression contains an error, the error is detected and reported by the compiler. This approach is known as ***static binding***.

However, if an expression is a dynamic expression (i.e. has the type `dynamic`) this indicates that any binding that it participates in should be based on its run-time type (i.e. the actual type of the object it denotes at run-time) rather than the type it has at compile-time. The binding of such an operation is therefore deferred until the time where the operation is to be executed during the running of the program. This is referred to as ***dynamic binding***.

When an operation is dynamically bound, little or no checking is performed by the compiler. Instead if the run-time binding fails, errors are reported as exceptions at run-time.

The following operations in C# are subject to binding:

- Member access: `e.M`
- Method invocation: `e.M(e1, ..., eN)`
- Delegate invocation: `e(e1, ..., eN)`
- Element access: `e[e1, ..., eN]`
- Object creation: `new C(e1, ..., eN)`
- Overloaded unary operators: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
- Overloaded binary operators: `+`, `-`, `*`, `/`, `%`, `&`, `&&`, `|`, `||`, `??`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- Implicit and explicit conversions

When no dynamic expressions are involved, C# defaults to static binding, which means that the compile-time types of constituent expressions are used in the selection process. However, when one of the constituent expressions in the operations listed above is a dynamic expression, the operation is instead dynamically bound.

Binding-time

Static binding takes place at compile-time, whereas dynamic binding takes place at run-time. In the following sections, the term ***binding-time*** refers to either compile-time or run-time, depending on when the binding takes place.

The following example illustrates the notions of static and dynamic binding and of binding-time:

```
object  o = 5;
dynamic d = 5;

Console.WriteLine(5);  // static  binding to Console.WriteLine(int)
Console.WriteLine(o);  // static  binding to Console.WriteLine(object)
Console.WriteLine(d);  // dynamic binding to Console.WriteLine(int)
```

The first two calls are statically bound: the overload of `Console.WriteLine` is picked based on the compile-time type of their argument. Thus, the binding-time is compile-time.

The third call is dynamically bound: the overload of `Console.WriteLine` is picked based on the run-time type of its argument. This happens because the argument is a dynamic expression -- its compile-time type is `dynamic`. Thus, the binding-time for the third call is run-time.

Dynamic binding

The purpose of dynamic binding is to allow C# programs to interact with ***dynamic objects***, i.e. objects that do not follow the normal rules of the C# type system. Dynamic objects may be objects from other programming languages with different types systems, or they may be objects that are programmatically setup to implement their own binding semantics for different operations.

The mechanism by which a dynamic object implements its own semantics is implementation defined. A given interface -- again implementation defined -- is implemented by dynamic objects to signal to the C# run-time that they have special semantics. Thus, whenever operations on a dynamic object are dynamically bound, their own binding semantics, rather than those of C# as specified in this document, take over.

While the purpose of dynamic binding is to allow interoperation with dynamic objects, C# allows dynamic binding on all objects, whether they are dynamic or not. This allows for a smoother integration of dynamic objects, as the results of operations on them may not themselves be dynamic objects, but are still of a type unknown to the programmer at compile-time. Also dynamic binding can help eliminate error-prone reflection-based code even when no objects involved are dynamic objects.

The following sections describe for each construct in the language exactly when dynamic binding is applied, what compile time checking -- if any -- is applied, and what the compile-time result and expression classification is.

Types of constituent expressions

When an operation is statically bound, the type of a constituent expression (e.g. a receiver, and argument, an index or an operand) is always considered to be the compile-time type of that expression.

When an operation is dynamically bound, the type of a constituent expression is determined in different ways depending on the compile-time type of the constituent expression:

- A constituent expression of compile-time type `dynamic` is considered to have the type of the actual value that the expression evaluates to at runtime
- A constituent expression whose compile-time type is a type parameter is considered to have the type which the type parameter is bound to at runtime

- Otherwise the constituent expression is considered to have its compile-time type.

Operators

Expressions are constructed from ***operands*** and ***operators***. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as `--x`) or postfix notation (such as `x++`).
- Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).
- Ternary operator. Only one ternary operator, `?:`, exists; it takes three operands and uses infix notation (`c ? x : y`).

The order of evaluation of operators in an expression is determined by the ***precedence*** and ***associativity*** of the operators (Operator precedence and associativity).

Operands in an expression are evaluated from left to right. For example, in `F(i) + G(i++) * H(i)`, method `F` is called using the old value of `i`, then method `G` is called with the old value of `i`, and, finally, method `H` is called with the new value of `i`. This is separate from and unrelated to operator precedence.

Certain operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (Operator overloading).

Operator precedence and associativity

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive_expression* consists of a sequence of *multiplicative_expression*s separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*`, `/`, and `%` operators.

The following table summarizes all operators in order of precedence from highest to lowest:

SECTION	CATEGORY	OPERATORS
Primary expressions	Primary	`x.y` `f(x)` `a[x]` `x++` `x--` `new` `typeof` `default` `checked` `unchecked` `delegate`
Unary operators	Unary	`+` `-` `!` `~` `++x` `--x` `(T)x`
Arithmetic operators	Multiplicative	`*` `/` `%`
Arithmetic operators	Additive	`+` `-`
Shift operators	Shift	`<<` `>>`
Relational and type-testing operators	Relational and type testing	`<` `>` `<=` `>=` `is` `as`
Relational and type-testing operators	Equality	`==` `!=`

SECTION	CATEGORY	OPERATORS
Logical operators	Logical AND	`&`
Logical operators	Logical XOR	`^`
Logical operators	Logical OR	`|`
Conditional logical operators	Conditional AND	`&&`
Conditional logical operators	Conditional OR	`||`
The null coalescing operator	Null coalescing	`??`
Conditional operator	Conditional	`?:`
Assignment operators, Anonymous function expressions	Assignment and lambda expression	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=` `=>`

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

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

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

Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including `operator` declarations in classes and structs (Operators). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered, as described in Unary operator overload resolution and Binary operator overload resolution.

The ***overloadable unary operators*** are:

```
+   -   !   ~   ++   --   true   false
```

Although `true` and `false` are not used explicitly in expressions (and therefore are not included in the precedence table in Operator precedence and associativity), they are considered operators because they are invoked in several expression contexts: boolean expressions (Boolean expressions) and expressions involving the conditional (Conditional operator), and conditional logical operators (Conditional logical operators).

The ***overloadable binary operators*** are:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the `=`, `&&`, `||`, `??`, `?:`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as`, and `is` operators.

When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded. For example, an overload of operator `*` is also an overload of operator `*=`. This is described further in Compound assignment. 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 (User-defined conversions).

Element access, such as `a[x]`, is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (Indexers).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix `++` and `--` operators. In the third entry, *op* denotes any overloadable binary operator.

OPERATOR NOTATION	FUNCTIONAL NOTATION
`op x`	`operator op(x)`
`x op`	`operator op(x)`
`x op y`	`operator op(x,y)`

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the `/` operator is always a binary operator, always has the precedence level specified in Operator precedence and associativity, and is always left-associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of `operator ==` should compare the two operands for equality and return an appropriate `bool` result.

The descriptions of individual operators in Primary expressions through Conditional logical operators specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms ***unary operator overload resolution***, ***binary operator overload resolution***, and ***numeric promotion***, definitions of which are found in the following sections.

Unary operator overload resolution

An operation of the form `op x` or `x op`, where `op` is an overloadable unary operator, and `x` is an expression of type `X`, is processed as follows:

- The set of candidate user-defined operators provided by `X` for the operation `operator op(x)` is determined using the rules of Candidate user-defined operators.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined unary `operator op` implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator

are specified in the description of the operator (Primary expressions and Unary operators).

- The overload resolution rules of Overload resolution are applied to the set of candidate operators to select the best operator with respect to the argument list `(x)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

Binary operator overload resolution

An operation of the form `x op y`, where `op` is an overloadable binary operator, `x` is an expression of type `X`, and `y` is an expression of type `Y`, is processed as follows:

- The set of candidate user-defined operators provided by `X` and `Y` for the operation `operator op(x,y)` is determined. The set consists of the union of the candidate operators provided by `X` and the candidate operators provided by `Y`, each determined using the rules of Candidate user-defined operators. If `X` and `Y` are the same type, or if `X` and `Y` are derived from a common base type, then shared candidate operators only occur in the combined set once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator op` implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (Arithmetic operators through Conditional logical operators). For predefined enum and delegate operators, the only operators considered are those defined by an enum or delegate type that is the binding-time type of one of the operands.
- The overload resolution rules of Overload resolution are applied to the set of candidate operators to select the best operator with respect to the argument list `(x,y)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

Candidate user-defined operators

Given a type `T` and an operation `operator op(A)`, where `op` is an overloadable operator and `A` is an argument list, the set of candidate user-defined operators provided by `T` for `operator op(A)` is determined as follows:

- Determine the type `T0`. If `T` is a nullable type, `T0` is its underlying type, otherwise `T0` is equal to `T`.
- For all `operator op` declarations in `T0` and all lifted forms of such operators, if at least one operator is applicable (Applicable function member) with respect to the argument list `A`, then the set of candidate operators consists of all such applicable operators in `T0`.
- Otherwise, if `T0` is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by `T0` is the set of candidate operators provided by the direct base class of `T0`, or the effective base class of `T0` if `T0` is a type parameter.

Numeric promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary `*` operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules (Overload resolution) are applied to this set of operators, the effect is to select the

first of the operators for which implicit conversions exist from the operand types. For example, for the operation `b * s`, where `b` is a `byte` and `s` is a `short`, overload resolution selects `operator *(int,int)` as the best operator. Thus, the effect is that `b` and `s` are converted to `int`, and the type of the result is `int`. Likewise, for the operation `i * d`, where `i` is an `int` and `d` is a `double`, overload resolution selects `operator *(double,double)` as the best operator.

Unary numeric promotions

Unary numeric promotion occurs for the operands of the predefined `+`, `-`, and `~` unary operators. Unary numeric promotion simply consists of converting operands of type `sbyte`, `byte`, `short`, `ushort`, or `char` to type `int`. Additionally, for the unary `-` operator, unary numeric promotion converts operands of type `uint` to type `long`.

Binary numeric promotions

Binary numeric promotion occurs for the operands of the predefined `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=`, and `<=` binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rules, in the order they appear here:

- If either operand is of type `decimal`, the other operand is converted to type `decimal`, or a binding-time error occurs if the other operand is of type `float` or `double`.
- Otherwise, if either operand is of type `double`, the other operand is converted to type `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to type `float`.
- Otherwise, if either operand is of type `ulong`, the other operand is converted to type `ulong`, or a binding-time error occurs if the other operand is of type `sbyte`, `short`, `int`, or `long`.
- Otherwise, if either operand is of type `long`, the other operand is converted to type `long`.
- Otherwise, if either operand is of type `uint` and the other operand is of type `sbyte`, `short`, or `int`, both operands are converted to type `long`.
- Otherwise, if either operand is of type `uint`, the other operand is converted to type `uint`.
- Otherwise, both operands are converted to type `int`.

Note that the first rule disallows any operations that mix the `decimal` type with the `double` and `float` types. The rule follows from the fact that there are no implicit conversions between the `decimal` type and the `double` and `float` types.

Also note that it is not possible for an operand to be of type `ulong` when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of `ulong` as well as the signed integral types.

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

In the example

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

a binding-time error occurs because a `decimal` cannot be multiplied by a `double`. The error is resolved by explicitly converting the second operand to `decimal`, as follows:

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

Lifted operators

Lifted operators permit predefined and user-defined operators that operate on non-nullable value types to also be used with nullable forms of those types. Lifted operators are constructed from predefined and user-defined operators that meet certain requirements, as described in the following:

- For the unary operators

  ```
  + ++ - -- ! ~
  ```

  a lifted form of an operator exists if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single `?` modifier to the operand and result types. The lifted operator produces a null value if the operand is null. Otherwise, the lifted operator unwraps the operand, applies the underlying operator, and wraps the result.

- For the binary operators

  ```
  + - * / % & | ^ << >>
  ```

  a lifted form of an operator exists if the operand and result types are all non-nullable value types. The lifted form is constructed by adding a single `?` modifier to each operand and result type. The lifted operator produces a null value if one or both operands are null (an exception being the `&` and `|` operators of the `bool?` type, as described in Boolean logical operators). Otherwise, the lifted operator unwraps the operands, applies the underlying operator, and wraps the result.

- For the equality operators

  ```
  == !=
  ```

  a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The lifted operator considers two null values equal, and a null value unequal to any non-null value. If both operands are non-null, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

- For the relational operators

  ```
  < > <= >=
  ```

  a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The lifted operator produces the value `false` if one or both operands are null. Otherwise, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

Member lookup

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup can occur as part of evaluating a *simple_name* (Simple names) or a *member_access* (Member access) in an expression. If the *simple_name* or *member_access* occurs as the *primary_expression* of an *invocation_expression* (Method invocations), the member is said to be invoked.

If a member is a method or event, or if it is a constant, field or property of either a delegate type (Delegates) or the type `dynamic` (The dynamic type), then the member is said to be *invocable*.

Member lookup considers not only the name of a member but also the number of type parameters the member has and whether the member is accessible. For the purposes of member lookup, generic methods and nested generic types have the number of type parameters indicated in their respective declarations and all other members have zero type parameters.

A member lookup of a name `N` with `K` type parameters in a type `T` is processed as follows:

- First, a set of accessible members named `N` is determined:
  - If `T` is a type parameter, then the set is the union of the sets of accessible members named `N` in each of the types specified as a primary constraint or secondary constraint (Type parameter constraints) for `T`, along with the set of accessible members named `N` in `object`.
  - Otherwise, the set consists of all accessible (Member access) members named `N` in `T`, including inherited members and the accessible members named `N` in `object`. If `T` is a constructed type, the set of members is obtained by substituting type arguments as described in Members of constructed types. Members that include an `override` modifier are excluded from the set.
- Next, if `K` is zero, all nested types whose declarations include type parameters are removed. If `K` is not zero, all members with a different number of type parameters are removed. Note that when `K` is zero, methods having type parameters are not removed, since the type inference process (Type inference) might be able to infer the type arguments.
- Next, if the member is *invoked*, all non-*invocable* members are removed from the set.
- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied:
  - If `M` is a constant, field, property, event, or enumeration member, then all members declared in a base type of `S` are removed from the set.
  - If `M` is a type declaration, then all non-types declared in a base type of `S` are removed from the set, and all type declarations with the same number of type parameters as `M` declared in a base type of `S` are removed from the set.
  - If `M` is a method, then all non-method members declared in a base type of `S` are removed from the set.
- Next, interface members that are hidden by class members are removed from the set. This step only has an effect if `T` is a type parameter and `T` has both an effective base class other than `object` and a non-empty effective interface set (Type parameter constraints). For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied if `S` is a class declaration other than `object`:
  - If `M` is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
  - If `M` is a method, then all non-method members declared in an interface declaration are removed from the set, and all methods with the same signature as `M` declared in an interface declaration are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined:
  - If the set consists of a single member that is not a method, then this member is the result of the lookup.
  - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
  - Otherwise, the lookup is ambiguous, and a binding-time error occurs.

For member lookups in types other than type parameters and interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in Interface member access.

Base types

For purposes of member lookup, a type `T` is considered to have the following base types:

- If `T` is `object`, then `T` has no base type.
- If `T` is an *enum_type*, the base types of `T` are the class types `System.Enum`, `System.ValueType`, and `object`.
- If `T` is a *struct_type*, the base types of `T` are the class types `System.ValueType` and `object`.
- If `T` is a *class_type*, the base types of `T` are the base classes of `T`, including the class type `object`.
- If `T` is an *interface_type*, the base types of `T` are the base interfaces of `T` and the class type `object`.
- If `T` is an *array_type*, the base types of `T` are the class types `System.Array` and `object`.
- If `T` is a *delegate_type*, the base types of `T` are the class types `System.Delegate` and `object`.

Function members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Destructors

Except for destructors and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list (Argument lists) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of generic methods may employ type inference to determine the set of type arguments to pass to the method. This process is described in Type inference.

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 Overload resolution.

Once a particular function member has been identified at binding-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in Compile-time checking of dynamic overload resolution.

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`.

CONSTRUCT	EXAMPLE	DESCRIPTION
	`T.F(x,y)`	Overload resolution is applied to select the best method `F` in the class or struct `T`. A binding-time error occurs if the method is not `static`. The method is invoked with the argument list `(x,y)`.
	`e.F(x,y)`	Overload resolution is applied to select the best method F in the class, struct, or interface given by the type of `e`. A binding-time error occurs if the method is `static`. The method is invoked with the instance expression `e` and the argument list `(x,y)`.
Property access	`P`	The `get` accessor of the property `P` in the containing class or struct is invoked. A compile-time error occurs if `P` is write-only. If `P` is not `static`, the instance expression is `this`.
	`P = value`	The `set` accessor of the property `P` in the containing class or struct is invoked with the argument list `(value)`. A compile-time error occurs if `P` is read-only. If `P` is not `static`, the instance expression is `this`.
	`T.P`	The `get` accessor of the property `P` in the class or struct `T` is invoked. A compile-time error occurs if `P` is not `static` or if `P` is write-only.
	`T.P = value`	The `set` accessor of the property `P` in the class or struct `T` is invoked with the argument list `(value)`. A compile-time error occurs if `P` is not `static` or if `P` is read-only.
	`e.P`	The `get` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `P` is `static` or if `P` is write-only.
	`e.P = value`	The `set` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e` and the argument list `(value)`. A binding-time error occurs if `P` is `static` or if `P` is read-only.

CONSTRUCT	EXAMPLE	DESCRIPTION
Event access	`E += value`	The `add` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
	`E -= value`	The `remove` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
	`T.E += value`	The `add` accessor of the event `E` in the class or struct `T` is invoked. A binding-time error occurs if `E` is not static.
	`T.E -= value`	The `remove` accessor of the event `E` in the class or struct `T` is invoked. A binding-time error occurs if `E` is not static.
	`e.E += value`	The `add` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `E` is static.
	`e.E -= value`	The `remove` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `E` is static.
Indexer access	`e[x,y]`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of e. The `get` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x,y)`. A binding-time error occurs if the indexer is write-only.
	`e[x,y] = value`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of `e`. The `set` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x,y,value)`. A binding-time error occurs if the indexer is read-only.
Operator invocation	`-x`	Overload resolution is applied to select the best unary operator in the class or struct given by the type of `x`. The selected operator is invoked with the argument list `(x)`.

CONSTRUCT	EXAMPLE	DESCRIPTION
	`x + y`	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of `x` and `y`. The selected operator is invoked with the argument list `(x,y)`.
Instance constructor invocation	`new T(x,y)`	Overload resolution is applied to select the best instance constructor in the class or struct `T`. The instance constructor is invoked with the argument list `(x,y)`.

Argument lists

Every function member and delegate invocation includes an argument list which provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For instance constructors, methods, indexers and delegates, the arguments are specified as an *argument_list*, as described below. For indexers, when invoking the `set` accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator.
- For properties, the argument list is empty when invoking the `get` accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the `set` accessor.
- For events, the argument list consists of the expression specified as the right operand of the `+=` or `-=` operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

The arguments of properties (Properties), events (Events), and user-defined operators (Operators) are always passed as value parameters (Value parameters). The arguments of indexers (Indexers) are always passed as value parameters (Value parameters) or parameter arrays (Parameter arrays). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, indexer or delegate invocation are specified as an *argument_list*:

```
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'ref' variable_reference
    | 'out' variable_reference
    ;
```

An *argument_list* consists of one or more *argument*s, separated by commas. Each argument consists of an optional *argument_name* followed by an *argument_value*. An *argument* with an *argument_name* is referred to as

a ***named argument***, whereas an *argument* without an *argument_name* is a ***positional argument***. It is an error for a positional argument to appear after a named argument in an *argument_list*.

The *argument_value* can take one of the following forms:

- An *expression*, indicating that the argument is passed as a value parameter (Value parameters).
- The keyword `ref` followed by a *variable_reference* (Variable references), indicating that the argument is passed as a reference parameter (Reference parameters). A variable must be definitely assigned (Definite assignment) before it can be passed as a reference parameter. The keyword `out` followed by a *variable_reference* (Variable references), indicating that the argument is passed as an output parameter (Output parameters). A variable is considered definitely assigned (Definite assignment) following a function member invocation in which the variable is passed as an output parameter.

Corresponding parameters

For each argument in an argument list there has to be a corresponding parameter in the function member or delegate being invoked.

The parameter list used in the following is determined as follows:

- For virtual methods and indexers defined in classes, the parameter list is picked from the most specific declaration or override of the function member, starting with the static type of the receiver, and searching through its base classes.
- For interface methods and indexers, the parameter list is picked form the most specific definition of the member, starting with the interface type and searching through the base interfaces. If no unique parameter list is found, a parameter list with inaccessible names and no optional parameters is constructed, so that invocations cannot use named parameters or omit optional arguments.
- For partial methods, the parameter list of the defining partial method declaration is used.
- For all other function members and delegates there is only a single parameter list, which is the one used.

The position of an argument or parameter is defined as the number of arguments or parameters preceding it in the argument list or parameter list.

The corresponding parameters for function member arguments are established as follows:

- Arguments in the *argument_list* of instance constructors, methods, indexers and delegates:
  - A positional argument where a fixed parameter occurs at the same position in the parameter list corresponds to that parameter.
  - A positional argument of a function member with a parameter array invoked in its normal form corresponds to the parameter array, which must occur at the same position in the parameter list.
  - A positional argument of a function member with a parameter array invoked in its expanded form, where no fixed parameter occurs at the same position in the parameter list, corresponds to an element in the parameter array.
  - A named argument corresponds to the parameter of the same name in the parameter list.
  - For indexers, when invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For properties, when invoking the `get` accessor there are no arguments. When invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For user-defined unary operators (including conversions), the single operand corresponds to the single parameter of the operator declaration.
- For user-defined binary operators, the left operand corresponds to the first parameter, and the right operand corresponds to the second parameter of the operator declaration.

Run-time evaluation of argument lists

During the run-time processing of a function member invocation (Compile-time checking of dynamic overload resolution), 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 (Implicit conversions) 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 (Parameter arrays). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (Applicable function member):

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression that is implicitly convertible (Implicit conversions) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more positional arguments for the parameter array, where each argument is an expression that is implicitly convertible (Implicit conversions) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Thus, the example

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

produces the output

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

The array co-variance rules (Array covariance) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, when an array element of a *reference_type* is passed as a reference or output parameter, a run-time check is required to ensure that the actual element type of the array is identical to that of the parameter. In the example

```
class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);        // Ok
        F(ref b[1]);        // ArrayTypeMismatchException
    }
}
```

the second invocation of `F` causes a `System.ArrayTypeMismatchException` to be thrown because the actual element type of `b` is `string` and not `object`.

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (Array creation expressions) was inserted around the expanded parameters. For example, given the declaration

```
void F(int x, int y, params object[] args);
```

the following invocations of the expanded form of the method

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspond exactly to

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

In particular, note that an empty array is created when there are zero arguments given for the parameter array.

When arguments are omitted from a function member with corresponding optional parameters, the default arguments of the function member declaration are implicitly passed. Because these are always constant, their evaluation will not impact the evaluation order of the remaining arguments.

Type inference

When a generic method is called without specifying type arguments, a ***type inference*** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration:

```
class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}
```

it is possible to invoke the `Choose` method without explicitly specifying a type argument:

```
int i = Chooser.Choose(5, 213);                // Calls Choose<int>

string s = Chooser.Choose("foo", "bar");       // Calls Choose<string>
```

Through type inference, the type arguments `int` and `string` are determined from the arguments to the method.

Type inference occurs as part of the binding-time processing of a method invocation (Method invocations) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a binding-time error. However, it often leads to a binding-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

```
Tr M<X1,...,Xn>(T1 x1, ..., Tm xm)
```

With a method call of the form `M(E1...Em)` the task of type inference is to find unique type arguments `S1...Sn` for each of the type parameters `X1...Xn` so that the call `M<S1...Sn>(E1...Em)` becomes valid.

During the process of inference each type parameter `Xi` is either *fixed* to a particular type `Si` or *unfixed* with an associated set of *bounds*. Each of the bounds is some type `T`. Initially each type variable `Xi` is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

Note: Type inference takes place not only when a generic method is called. Type inference for conversion of method groups is described in Type inference for conversion of method groups and finding the best common type of a set of expressions is described in Finding the best common type of a set of expressions.

The first phase

For each of the method arguments `Ei`:

- If `Ei` is an anonymous function, an *explicit parameter type inference* (Explicit parameter type inferences) is made from `Ei` to `Ti`
- Otherwise, if `Ei` has a type `U` and `xi` is a value parameter then a *lower-bound inference* is made *from* `U` *to* `Ti`.
- Otherwise, if `Ei` has a type `U` and `xi` is a `ref` or `out` parameter then an *exact inference* is made *from* `U` *to* `Ti`.
- Otherwise, no inference is made for this argument.

The second phase

The second phase proceeds as follows:

- All *unfixed* type variables `Xi` which do not *depend on* (Dependence) any `Xj` are fixed (Fixing).
- If no such type variables exist, all *unfixed* type variables `Xi` are *fixed* for which all of the following hold:
  - There is at least one type variable `Xj` that depends on `Xi`

- `Xi` has a non-empty set of bounds

- If no such type variables exist and there are still *unfixed* type variables, type inference fails.
- Otherwise, if no further *unfixed* type variables exist, type inference succeeds.
- Otherwise, for all arguments `Ei` with corresponding parameter type `Ti` where the *output types* (Output types) contain *unfixed* type variables `Xj` but the *input types* (Input types) do not, an *output type inference* (Output type inferences) is made *from* `Ei` *to* `Ti`. Then the second phase is repeated.

Input types

If `E` is a method group or implicitly typed anonymous function and `T` is a delegate type or expression tree type then all the parameter types of `T` are *input types* of `E` *with type* `T`.

Output types

If `E` is a method group or an anonymous function and `T` is a delegate type or expression tree type then the return type of `T` is an *output type of* `E` *with type* `T`.

Dependence

An *unfixed* type variable `Xi` *depends directly on* an unfixed type variable `Xj` if for some argument `Ek` with type `Tk` `Xj` occurs in an *input type* of `Ek` with type `Tk` and `Xi` occurs in an *output type* of `Ek` with type `Tk`.

`Xj` *depends on* `Xi` if `Xj` *depends directly on* `Xi` or if `Xi` *depends directly on* `Xk` and `Xk` *depends on* `Xj`. Thus "depends on" is the transitive but not reflexive closure of "depends directly on".

Output type inferences

An *output type inference* is made *from* an expression `E` *to* a type `T` in the following way:

- If `E` is an anonymous function with inferred return type `U` (Inferred return type) and `T` is a delegate type or expression tree type with return type `Tb`, then a *lower-bound inference* (Lower-bound inferences) is made *from* `U` *to* `Tb`.
- Otherwise, if `E` is a method group and `T` is a delegate type or expression tree type with parameter types `T1...Tk` and return type `Tb`, and overload resolution of `E` with the types `T1...Tk` yields a single method with return type `U`, then a *lower-bound inference* is made *from* `U` *to* `Tb`.
- Otherwise, if `E` is an expression with type `U`, then a *lower-bound inference* is made *from* `U` *to* `T`.
- Otherwise, no inferences are made.

Explicit parameter type inferences

An *explicit parameter type inference* is made *from* an expression `E` *to* a type `T` in the following way:

- If `E` is an explicitly typed anonymous function with parameter types `U1...Uk` and `T` is a delegate type or expression tree type with parameter types `V1...Vk` then for each `Ui` an *exact inference* (Exact inferences) is made *from* `Ui` *to* the corresponding `Vi`.

Exact inferences

An *exact inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* `Xi` then `U` is added to the set of exact bounds for `Xi`.
- Otherwise, sets `V1...Vk` and `U1...Uk` are determined by checking if any of the following cases apply:
  - `V` is an array type `V1[...]` and `U` is an array type `U1[...]` of the same rank
  - `V` is the type `V1?` and `U` is the type `U1?`
  - `V` is a constructed type `C<V1...Vk>` and `U` is a constructed type `C<U1...Uk>`

  If any of these cases apply then an *exact inference* is made *from* each `Ui` *to* the corresponding `Vi`.
- Otherwise no inferences are made.

Lower-bound inferences

A *lower-bound inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* `Xi` then `U` is added to the set of lower bounds for `Xi`.
- Otherwise, if `V` is the type `V1?` and `U` is the type `U1?` then a lower bound inference is made from `U1` to `V1`.
- Otherwise, sets `U1...Uk` and `V1...Vk` are determined by checking if any of the following cases apply:
  - `V` is an array type `V1[...]` and `U` is an array type `U1[...]` (or a type parameter whose effective base type is `U1[...]`) of the same rank
  - `V` is one of `IEnumerable<V1>`, `ICollection<V1>` or `IList<V1>` and `U` is a one-dimensional array type `U1[]` (or a type parameter whose effective base type is `U1[]`)
  - `V` is a constructed class, struct, interface or delegate type `C<V1...Vk>` and there is a unique type `C<U1...Uk>` such that `U` (or, if `U` is a type parameter, its effective base class or any member of its effective interface set) is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) `C<U1...Uk>`.

    (The "uniqueness" restriction means that in the case interface `C<T> {} class U: C<X>, C<Y> {}`, then no inference is made when inferring from `U` to `C<T>` because `U1` could be `X` or `Y`.)

  If any of these cases apply then an inference is made *from* each `Ui` *to* the corresponding `Vi` as follows:

  - If `Ui` is not known to be a reference type then an *exact inference* is made
  - Otherwise, if `U` is an array type then a *lower-bound inference* is made
  - Otherwise, if `V` is `C<V1...Vk>` then inference depends on the i-th type parameter of `C`:
    - If it is covariant then a *lower-bound inference* is made.
    - If it is contravariant then an *upper-bound inference* is made.
    - If it is invariant then an *exact inference* is made.
- Otherwise, no inferences are made.

Upper-bound inferences

An *upper-bound inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* `Xi` then `U` is added to the set of upper bounds for `Xi`.
- Otherwise, sets `V1...Vk` and `U1...Uk` are determined by checking if any of the following cases apply:
  - `U` is an array type `U1[...]` and `V` is an array type `V1[...]` of the same rank
  - `U` is one of `IEnumerable<Ue>`, `ICollection<Ue>` or `IList<Ue>` and `V` is a one-dimensional array type `Ve[]`
  - `U` is the type `U1?` and `V` is the type `V1?`
  - `U` is constructed class, struct, interface or delegate type `C<U1...Uk>` and `V` is a class, struct, interface or delegate type which is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) a unique type `C<V1...Vk>`

    (The "uniqueness" restriction means that if we have `interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}`, then no inference is made when inferring from `C<U1>` to `V<Q>`. Inferences are not made from `U1` to either `X<Q>` or `Y<Q>`.)

  If any of these cases apply then an inference is made *from* each `Ui` *to* the corresponding `Vi` as follows:

  - If `Ui` is not known to be a reference type then an *exact inference* is made
  - Otherwise, if `V` is an array type then an *upper-bound inference* is made
  - Otherwise, if `U` is `C<U1...Uk>` then inference depends on the i-th type parameter of `C`:
    - If it is covariant then an *upper-bound inference* is made.
    - If it is contravariant then a *lower-bound inference* is made.

    - If it is invariant then an *exact inference* is made.
- Otherwise, no inferences are made.

Fixing

An *unfixed* type variable `Xi` with a set of bounds is *fixed* as follows:

- The set of *candidate types* `Uj` starts out as the set of all types in the set of bounds for `Xi`.
- We then examine each bound for `Xi` in turn: For each exact bound `U` of `Xi` all types `Uj` which are not identical to `U` are removed from the candidate set. For each lower bound `U` of `Xi` all types `Uj` to which there is *not* an implicit conversion from `U` are removed from the candidate set. For each upper bound `U` of `Xi` all types `Uj` from which there is *not* an implicit conversion to `U` are removed from the candidate set.
- If among the remaining candidate types `Uj` there is a unique type `V` from which there is an implicit conversion to all the other candidate types, then `Xi` is fixed to `V`.
- Otherwise, type inference fails.

Inferred return type

The inferred return type of an anonymous function `F` is used during type inference and overload resolution. The inferred return type can only be determined for an anonymous function where all parameter types are known, either because they are explicitly given, provided through an anonymous function conversion or inferred during type inference on an enclosing generic method invocation.

The ***inferred result type*** is determined as follows:

- If the body of `F` is an *expression* that has a type, then the inferred result type of `F` is the type of that expression.
- If the body of `F` is a *block* and the set of expressions in the block's `return` statements has a best common type `T` (Finding the best common type of a set of expressions), then the inferred result type of `F` is `T`.
- Otherwise, a result type cannot be inferred for `F`.

The ***inferred return type*** is determined as follows:

- If `F` is async and the body of `F` is either an expression classified as nothing (Expression classifications), or a statement block where no return statements have expressions, the inferred return type is `System.Threading.Tasks.Task`
- If `F` is async and has an inferred result type `T`, the inferred return type is `System.Threading.Tasks.Task<T>`.
- If `F` is non-async and has an inferred result type `T`, the inferred return type is `T`.
- Otherwise a return type cannot be inferred for `F`.

As an example of type inference involving anonymous functions, consider the `Select` extension method declared in the `System.Linq.Enumerable` class:

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource,TResult>(
            this IEnumerable<TSource> source,
            Func<TSource,TResult> selector)
        {
            foreach (TSource element in source) yield return selector(element);
        }
    }
}
```

Assuming the `System.Linq` namespace was imported with a `using` clause, and given a class `Customer` with a `Name` property of type `string`, the `Select` method can be used to select the names of a list of customers:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

The extension method invocation (Extension method invocations) of `Select` is processed by rewriting the invocation to a static method invocation:

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the `customers` argument is related to the `source` parameter, inferring `T` to be `Customer`. Then, using the anonymous function type inference process described above, `c` is given type `Customer`, and the expression `c.Name` is related to the return type of the `selector` parameter, inferring `S` to be `string`. Thus, the invocation is equivalent to

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

and the result is of type `IEnumerable<string>`.

The following example demonstrates how anonymous function type inference allows type information to "flow" between arguments in a generic method invocation. Given the method:

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
    return f2(f1(value));
}
```

Type inference for the invocation:

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

proceeds as follows: First, the argument `"1:15:30"` is related to the `value` parameter, inferring `X` to be `string`. Then, the parameter of the first anonymous function, `s`, is given the inferred type `string`, and the expression `TimeSpan.Parse(s)` is related to the return type of `f1`, inferring `Y` to be `System.TimeSpan`. Finally, the parameter of the second anonymous function, `t`, is given the inferred type `System.TimeSpan`, and the expression `t.TotalSeconds` is related to the return type of `f2`, inferring `Z` to be `double`. Thus, the result of the invocation is of type `double`.

Type inference for conversion of method groups

Similar to calls of generic methods, type inference must also be applied when a method group `M` containing a generic method is converted to a given delegate type `D` (Method group conversions). Given a method

```
Tr M<X1...Xn>(T1 x1 ... Tm xm)
```

and the method group `M` being assigned to the delegate type `D` the task of type inference is to find type arguments `S1...Sn` so that the expression:

```
M<S1...Sn>
```

becomes compatible (Delegate declarations) with `D`.

Unlike the type inference algorithm for generic method calls, in this case there are only argument *types*, no argument *expressions*. In particular, there are no anonymous functions and hence no need for multiple phases of

inference.

Instead, all `Xi` are considered *unfixed*, and a *lower-bound inference* is made *from* each argument type `Uj` of `D` *to* the corresponding parameter type `Tj` of `M`. If for any of the `Xi` no bounds were found, type inference fails. Otherwise, all `Xi` are *fixed* to corresponding `Si`, which are the result of type inference.

Finding the best common type of a set of expressions

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous functions with *block* bodies are found in this way.

Intuitively, given a set of expressions `E1...Em` this inference should be equivalent to calling a method

```
Tr M<X>(X x1 ... X xm)
```

with the `Ei` as arguments.

More precisely, the inference starts out with an *unfixed* type variable `X`. *Output type inferences* are then made *from* each `Ei` *to* `X`. Finally, `X` is *fixed* and, if successful, the resulting type `S` is the resulting best common type for the expressions. If no such `S` exists, the expressions have no best common type.

Overload resolution

Overload resolution is a binding-time mechanism for selecting the best function member to invoke given an argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an *invocation_expression* (Method invocations).
- Invocation of an instance constructor named in an *object_creation_expression* (Object creation expressions).
- Invocation of an indexer accessor through an *element_access* (Element access).
- Invocation of a predefined or user-defined operator referenced in an expression (Unary operator overload resolution and Binary operator overload resolution).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a method invocation does not include methods marked `override` (Member lookup), and methods in a base class are not candidates if any method in a derived class is applicable (Method invocations).

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- Given the set of applicable candidate function members, the best function member in that set is located. If the set contains only one function member, then that function member is the best function member. Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in Better function member. If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a binding-time error occurs.

The following sections define the exact meanings of the terms ***applicable function member*** and ***better function member***.

Applicable function member

A function member is said to be an ***applicable function member*** with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in Corresponding parameters, and any parameter to which no argument corresponds is an optional parameter.
- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to

the parameter passing mode of the corresponding parameter, and
  - for a value parameter or a parameter array, an implicit conversion (Implicit conversions) exists from the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its ***normal form***. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its ***expanded form***:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
  - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion (Implicit conversions) exists from the type of the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

Better function member

For the purposes of determining the better function member, a stripped-down argument list A is constructed containing just the argument expressions themselves in the order they appear in the original argument list.

Parameter lists for each of the candidate function members are constructed in the following way:

- The expanded form is used if the function member was applicable only in the expanded form.
- Optional parameters with no corresponding arguments are removed from the parameter list
- The parameters are reordered so that they occur at the same position as the corresponding argument in the argument list.

Given an argument list `A` with a set of argument expressions `{E1, E2, ..., En}` and two applicable function members `Mp` and `Mq` with parameter types `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}`, `Mp` is defined to be a ***better function member*** than `Mq` if

- for each argument, the implicit conversion from `Ex` to `Qx` is not better than the implicit conversion from `Ex` to `Px`, and
- for at least one argument, the conversion from `Ex` to `Px` is better than the conversion from `Ex` to `Qx`.

When performing this evaluation, if `Mp` or `Mq` is applicable in its expanded form, then `Px` or `Qx` refers to a parameter in the expanded form of the parameter list.

In case the parameter type sequences `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}` are equivalent (i.e. each `Pi` has an identity conversion to the corresponding `Qi`), the following tie-breaking rules are applied, in order, to determine the better function member.

- If `Mp` is a non-generic method and `Mq` is a generic method, then `Mp` is better than `Mq`.
- Otherwise, if `Mp` is applicable in its normal form and `Mq` has a `params` array and is applicable only in its expanded form, then `Mp` is better than `Mq`.
- Otherwise, if `Mp` has more declared parameters than `Mq`, then `Mp` is better than `Mq`. This can occur if both methods have `params` arrays and are applicable only in their expanded forms.

- Otherwise if all parameters of `Mp` have a corresponding argument whereas default arguments need to be substituted for at least one optional parameter in `Mq` then `Mp` is better than `Mq`.
- Otherwise, if `Mp` has more specific parameter types than `Mq`, then `Mp` is better than `Mq`. Let `{R1, R2, ..., Rn}` and `{S1, S2, ..., Sn}` represent the uninstantiated and unexpanded parameter types of `Mp` and `Mq`. `Mp`'s parameter types are more specific than `Mq`'s if, for each parameter, `Rx` is not less specific than `Sx`, and, for at least one parameter, `Rx` is more specific than `Sx`:
  - A type parameter is less specific than a non-type parameter.
  - Recursively, a constructed type is more specific than another constructed type (with the same number of type arguments) if at least one type argument is more specific and no type argument is less specific than the corresponding type argument in the other.
  - An array type is more specific than another array type (with the same number of dimensions) if the element type of the first is more specific than the element type of the second.
- Otherwise if one member is a non-lifted operator and the other is a lifted operator, the non-lifted one is better.
- Otherwise, neither function member is better.

Better conversion from expression

Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if `E` does not exactly match `T2` and at least one of the following holds:

- `E` exactly matches `T1` (Exactly matching Expression)
- `T1` is a better conversion target than `T2` (Better conversion target)

Exactly matching Expression

Given an expression `E` and a type `T`, `E` exactly matches `T` if one of the following holds:

- `E` has a type `S`, and an identity conversion exists from `S` to `T`
- `E` is an anonymous function, `T` is either a delegate type `D` or an expression tree type `Expression<D>` and one of the following holds:
  - An inferred return type `X` exists for `E` in the context of the parameter list of `D` (Inferred return type), and an identity conversion exists from `X` to the return type of `D`
  - Either `E` is non-async and `D` has a return type `Y` or `E` is async and `D` has a return type `Task<Y>`, and one of the following holds:
    - The body of `E` is an expression that exactly matches `Y`
    - The body of `E` is a statement block where every return statement returns an expression that exactly matches `Y`

Better conversion target

Given two different types `T1` and `T2`, `T1` is a better conversion target than `T2` if no implicit conversion from `T2` to `T1` exists, and at least one of the following holds:

- An implicit conversion from `T1` to `T2` exists
- `T1` is either a delegate type `D1` or an expression tree type `Expression<D1>`, `T2` is either a delegate type `D2` or an expression tree type `Expression<D2>`, `D1` has a return type `S1` and one of the following holds:
  - `D2` is void returning
  - `D2` has a return type `S2`, and `S1` is a better conversion target than `S2`
- `T1` is `Task<S1>`, `T2` is `Task<S2>`, and `S1` is a better conversion target than `S2`
- `T1` is `S1` or `S1?` where `S1` is a signed integral type, and `T2` is `S2` or `S2?` where `S2` is an unsigned integral type. Specifically:
  - `S1` is `sbyte` and `S2` is `byte`, `ushort`, `uint`, or `ulong`
  - `S1` is `short` and `S2` is `ushort`, `uint`, or `ulong`

- `S1` is `int` and `S2` is `uint`, or `ulong`
- `S1` is `long` and `S2` is `ulong`

Overloading in generic classes

While signatures as declared must be unique, it is possible that substitution of type arguments results in identical signatures. In such cases, the tie-breaking rules of overload resolution above will pick the most specific member.

The following examples show overloads that are valid and invalid according to this rule:

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);                  // Overload resolution for G<int>.F1
    int F1(int i);                // will pick non-generic

    void F2(I1<U> a);             // Valid overload
    void F2(I2<U> a);
}

class G2<U,V>
{
    void F3(U u, V v);            // Valid, but overload resolution for
    void F3(V v, U u);            // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);        // Valid, but overload resolution for
    void F4(I1<V> v, U u);        // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2);      // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);             // valid overload
    void F6(out V v);
}
```

Compile-time checking of dynamic overload resolution

For most dynamically bound operations the set of possible candidates for resolution is unknown at compile-time. In certain cases, however the candidate set is known at compile-time:

- Static method calls with dynamic arguments
- Instance method calls where the receiver is not a dynamic expression
- Indexer calls where the receiver is not a dynamic expression
- Constructor calls with dynamic arguments

In these cases a limited compile-time check is performed for each candidate to see if any of them could possibly apply at run-time.This check consists of the following steps:

- Partial type inference: Any type argument that does not depend directly or indirectly on an argument of type `dynamic` is inferred using the rules of Type inference. The remaining type arguments are unknown.
- Partial applicability check: Applicability is checked according to Applicable function member, but ignoring parameters whose types are unknown.
- If no candidate passes this test, a compile-time error occurs.

Function member invocation

This section describes the process that takes place at run-time to invoke a particular function member. It is assumed that a binding-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are instance constructors, static methods, static property accessors, and user-defined operators. Static function members are always non-virtual.
- Instance function members. These are instance methods, instance property accessors, and indexer accessors. Instance function members are either non-virtual or virtual, and are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as `this` (This access).

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 Argument lists.
  - `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 Argument lists.
  - `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 Argument lists.
  - If the type of `E` is a *value_type*, a boxing conversion (Boxing conversions) is performed to convert `E` to type `object`, and `E` is considered to be of type `object` in the following steps. In this case, `M` could only be a member of `System.Object`.
  - The value of `E` is checked to be valid. If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - The function member implementation to invoke is determined:
    - If the binding-time type of `E` is an interface, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the interface mapping rules (Interface mapping) 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 (Virtual methods) of `M` with respect to the run-time type of the instance referenced by `E`.
    - Otherwise, `M` is a non-virtual function member, and the function member to invoke is `M` itself.
  - The function member implementation determined in the step above is invoked. The object referenced by `E` becomes the object referenced by `this`.

Invocations on boxed instances

A function member implemented in a *value_type* can be invoked through a boxed instance of that *value_type* in the following situations:

- When the function member is an `override` of a method inherited from type `object` and is invoked through an instance expression of type `object`.
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface_type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value_type*, and this variable becomes the variable referenced by `this` within the function member invocation. In particular, this means that when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance.

Primary expressions

Primary expressions include the simplest forms of expressions.

```
primary_expression
    : primary_no_array_creation_expression
    | array_creation_expression
    ;

primary_no_array_creation_expression
    : literal
    | interpolated_string_expression
    | simple_name
    | parenthesized_expression
    | member_access
    | invocation_expression
    | element_access
    | this_access
    | base_access
    | post_increment_expression
    | post_decrement_expression
    | object_creation_expression
    | delegate_creation_expression
    | anonymous_object_creation_expression
    | typeof_expression
    | checked_expression
    | unchecked_expression
    | default_value_expression
    | nameof_expression
    | anonymous_method_expression
    | primary_no_array_creation_expression_unsafe
    ;
```

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];
```

Literals

A *primary_expression* that consists of a *literal* (Literals) is classified as a value.

Interpolated strings

An *interpolated_string_expression* consists of a `$` sign followed by a regular or verbatim string literal, wherein holes, delimited by `{` and `}`, enclose expressions and formatting specifications. An interpolated string expression is the result of an *interpolated_string_literal* that has been broken up into individual tokens, as described in Interpolated string literals.

```
interpolated_string_expression
    : '$' interpolated_regular_string
    | '$' interpolated_verbatim_string
    ;

interpolated_regular_string
    : interpolated_regular_string_whole
    | interpolated_regular_string_start interpolated_regular_string_body interpolated_regular_string_end
    ;

interpolated_regular_string_body
    : interpolation (interpolated_regular_string_mid interpolation)*
    ;

interpolation
    : expression
    | expression ',' constant_expression
    ;

interpolated_verbatim_string
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start interpolated_verbatim_string_body interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_body
    : interpolation (interpolated_verbatim_string_mid interpolation)+
    ;
```

The *constant_expression* in an interpolation must have an implicit conversion to `int`.

An *interpolated_string_expression* is classified as a value. If it is immediately converted to `System.IFormattable` or `System.FormattableString` with an implicit interpolated string conversion (Implicit interpolated string conversions), the interpolated string expression has that type. Otherwise, it has the type `string`.

If the type of an interpolated string is `System.IFormattable` or `System.FormattableString`, the meaning is a call to `System.Runtime.CompilerServices.FormattableStringFactory.Create`. If the type is `string`, the meaning of the expression is a call to `string.Format`. In both cases, the argument list of the call consists of a format string literal with placeholders for each interpolation, and an argument for each expression corresponding to the place holders.

The format string literal is constructed as follows, where `N` is the number of interpolations in the *interpolated_string_expression*:

- If an *interpolated_regular_string_whole* or an *interpolated_verbatim_string_whole* follows the `$` sign, then the format string literal is that token.
- Otherwise, the format string literal consists of:
  - First the *interpolated_regular_string_start* or *interpolated_verbatim_string_start*
  - Then for each number `I` from `0` to `N-1`:
    - The decimal representation of `I`
    - Then, if the corresponding *interpolation* has a *constant_expression*, a `,` (comma) followed by the decimal representation of the value of the *constant_expression*
    - Then the *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_mid* or *interpolated_verbatim_string_end* immediately following the corresponding interpolation.

The subsequent arguments are simply the *expressions* from the *interpolations* (if any), in order.

TODO: examples.

Simple names

A *simple_name* consists of an identifier, optionally followed by a type argument list:

```
simple_name
    : identifier type_argument_list?
    ;
```

A *simple_name* is either of the form `I` or of the form `I<A1,...,Ak>`, where `I` is a single identifier and `<A1,...,Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `K` to be zero. The *simple_name* is evaluated and classified as follows:

- If `K` is zero and the *simple_name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable declaration space (Declarations) contains a local variable, parameter or constant with name `I`, then the *simple_name* refers to that local variable, parameter or constant and is classified as a variable or value.
- If `K` is zero and the *simple_name* appears within the body of a generic method declaration and if that declaration includes a type parameter with name `I`, then the *simple_name* refers to that type parameter.
- Otherwise, for each instance type `T` (The instance type), starting with the instance type of the immediately enclosing type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
  - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *simple_name* refers to that type parameter.
  - Otherwise, if a member lookup (Member lookup) of `I` in `T` with `K` type arguments produces a match:
    - If `T` is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`. If a type argument list was specified, it is used in calling a generic method (Method invocations).
    - Otherwise, if `T` is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the body of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (Member access) of the form `this.I`. This can only happen when `K` is zero.
    - Otherwise, the result is the same as a member access (Member access) of the form `T.I` or `T.I<A1,...,Ak>`. In this case, it is a binding-time error for the *simple_name* to refer to an instance member.
- Otherwise, for each namespace `N`, starting with the namespace in which the *simple_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If `K` is zero and `I` is the name of a namespace in `N`, then:
    - If the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *simple_name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *simple_name* refers to the namespace named `I` in `N`.
  - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
    - If `K` is zero and the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the

simple_name is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *simple_name* occurs is enclosed by a namespace declaration for `N`:
    - If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type, then the *simple_name* refers to that namespace or type.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_static_directive*s of the namespace declaration contain exactly one accessible type or non-extension static member having name `I` and `K` type parameters, then the *simple_name* refers to that type or member constructed with the given type arguments.
    - Otherwise, if the namespaces and types imported by the *using_namespace_directive*s of the namespace declaration contain more than one accessible type or non-extension-method static member having name `I` and `K` type parameters, then the *simple_name* is ambiguous and an error occurs.

  Note that this entire step is exactly parallel to the corresponding step in the processing of a *namespace_or_type_name* (Namespace and type names).

- Otherwise, the *simple_name* is undefined and a compile-time error occurs.

Parenthesized expressions

A *parenthesized_expression* consists of an *expression* enclosed in parentheses.

```
parenthesized_expression
    : '(' expression ')'
    ;
```

A *parenthesized_expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a namespace or type, a compile-time error occurs. Otherwise, the result of the *parenthesized_expression* is the result of the evaluation of the contained *expression*.

Member access

A *member_access* consists of a *primary_expression*, a *predefined_type*, or a *qualified_alias_member*, followed by a "`.`" token, followed by an *identifier*, optionally followed by a *type_argument_list*.

```
member_access
    : primary_expression '.' identifier type_argument_list?
    | predefined_type '.' identifier type_argument_list?
    | qualified_alias_member '.' identifier
    ;

predefined_type
    : 'bool'   | 'byte'  | 'char'  | 'decimal' | 'double' | 'float' | 'int' | 'long'
    | 'object' | 'sbyte' | 'short' | 'string'  | 'uint'   | 'ulong' | 'ushort'
    ;
```

The *qualified_alias_member* production is defined in Namespace alias qualifiers.

A *member_access* is either of the form `E.I` or of the form `E.I<A1, ..., Ak>`, where `E` is a primary-expression, `I` is a single identifier and `<A1, ..., Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `K` to be zero.

A *member_access* with a *primary_expression* of type `dynamic` is dynamically bound (Dynamic binding). In this case the compiler classifies the member access as a property access of type `dynamic`. The rules below to

determine the meaning of the *member_access* are then applied at run-time, using the run-time type instead of the compile-time type of the *primary_expression*. If this run-time classification leads to a method group, then the member access must be the *primary_expression* of an *invocation_expression*.

The *member_access* is evaluated and classified as follows:

- If `K` is zero and `E` is a namespace and `E` contains a nested namespace with name `I`, then the result is that namespace.
- Otherwise, if `E` is a namespace and `E` contains an accessible type having name `I` and `K` type parameters, then the result is that type constructed with the given type arguments.
- If `E` is a *predefined_type* or a *primary_expression* classified as a type, if `E` is not a type parameter, and if a member lookup (Member lookup) of `I` in `E` with `K` type parameters produces a match, then `E.I` is evaluated and classified as follows:
  - If `I` identifies a type, then the result is that type constructed with the given type arguments.
  - If `I` identifies one or more methods, then the result is a method group with no associated instance expression. If a type argument list was specified, it is used in calling a generic method (Method invocations).
  - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
  - If `I` identifies a `static` field:
    - If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field `I` in `E`.
    - Otherwise, the result is a variable, namely the static field `I` in `E`.
  - If `I` identifies a `static` event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event_accessor_declarations* (Events), then `E.I` is processed exactly as if `I` were a static field.
    - Otherwise, the result is an event access with no associated instance expression.
  - If `I` identifies a constant, then the result is a value, namely the value of that constant.
    - If `I` identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
    - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is a property access, indexer access, variable, or value, the type of which is `T`, and a member lookup (Member lookup) of `I` in `T` with `K` type arguments produces a match, then `E.I` is evaluated and classified as follows:
  - First, if `E` is a property or indexer access, then the value of the property or indexer access is obtained (Values of expressions) and `E` is reclassified as a value.
  - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`. If a type argument list was specified, it is used in calling a generic method (Method invocations).
  - If `I` identifies an instance property,
    - If `E` is `this`, `I` identifies an automatically implemented property (Automatically implemented properties) without a setter, and the reference occurs within an instance constructor for a class or struct type `T`, then the result is a variable, namely the hidden backing field for the auto-property given by `I` in the instance of `T` given by `this`.
    - Otherwise, the result is a property access with an associated instance expression of `E`.
  - If `T` is a *class_type* and `I` identifies an instance field of that *class_type*:
    - If the value of `E` is `null`, then a `System.NullReferenceException` is thrown.

    - Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field `I` in the object referenced by `E`.
    - Otherwise, the result is a variable, namely the field `I` in the object referenced by `E`.
  - If `T` is a *struct_type* and `I` identifies an instance field of that *struct_type*:
    - If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field `I` in the struct instance given by `E`.
    - Otherwise, the result is a variable, namely the field `I` in the struct instance given by `E`.
  - If `I` identifies an instance event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event_accessor_declarations* (Events), and the reference does not occur as the left-hand side of a `+=` or `-=` operator, then `E.I` is processed exactly as if `I` was an instance field.
    - Otherwise, the result is an event access with an associated instance expression of `E`.
- Otherwise, an attempt is made to process `E.I` as an extension method invocation (Extension method invocations). If this fails, `E.I` is an invalid member reference, and a binding-time error occurs.

Identical simple names and type names

In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a *simple_name* (Simple names) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as a *type_name* (Namespace and type names), then both possible meanings of `E` are permitted. The two possible meanings of `E.I` are never ambiguous, since `I` must necessarily be a member of the type `E` in both cases. In other words, the rule simply permits access to the static members and nested types of `E` where a compile-time error would otherwise have occurred. For example:

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
}

class A
{
    public Color Color;                // Field Color of type Color

    void F() {
        Color = Color.Black;           // References Color.Black static member
        Color = Color.Complement();    // Invokes Complement() on Color field
    }

    static void G() {
        Color c = Color.White;         // References Color.White static member
    }
}
```

Grammar ambiguities

The productions for *simple_name* (Simple names) and *member_access* (Member access) can give rise to ambiguities in the grammar for expressions. For example, the statement:

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

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted

as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple_name* (Simple names), *member_access* (Member access), or *pointer_member_access* (Pointer member access) ending with a *type_argument_list* (Type arguments), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type_argument_list* is retained as part of the *simple_name*, *member_access* or *pointer_member_access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type_argument_list* is not considered to be part of the *simple_name*, *member_access* or *pointer_member_access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type_argument_list* in a *namespace_or_type_name* (Namespace and type names). The statement

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

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

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

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

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

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

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace_or_type_name* with a *type_argument_list*.

Invocation expressions

An *invocation_expression* is used to invoke a method.

```
invocation_expression
    : primary_expression '(' argument_list? ')'
    ;
```

An *invocation_expression* is dynamically bound (Dynamic binding) if at least one of the following holds:

- The *primary_expression* has compile-time type `dynamic`.
- At least one argument of the optional *argument_list* has compile-time type `dynamic` and the *primary_expression* does not have a delegate type.

In this case the compiler classifies the *invocation_expression* as a value of type `dynamic`. The rules below to determine the meaning of the *invocation_expression* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary_expression* and arguments which have the compile-time type

`dynamic`. If the *primary_expression* does not have compile-time type `dynamic`, then the method invocation undergoes a limited compile time check as described in Compile-time checking of dynamic overload resolution.

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 (Method invocations). If the *primary_expression* is a value of a *delegate_type*, the *invocation_expression* is a delegate invocation (Delegate invocations). If the *primary_expression* is neither a method group nor a value of a *delegate_type*, a binding-time error occurs.

The optional *argument_list* (Argument lists) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation_expression* is classified as follows:

- If the *invocation_expression* invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing is permitted only in the context of a *statement_expression* (Expression statements) or as the body of a *lambda_expression* (Anonymous function expressions). Otherwise a binding-time error occurs.
- Otherwise, the result is a value of the type returned by the method or delegate.

Method invocations

For a method invocation, the *primary_expression* of the *invocation_expression* must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument_list*.

The binding-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type_argument_list*), and `A` is an optional *argument_list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`:
  - If `F` is non-generic, `F` is a candidate when:
    - `M` has no type argument list, and
    - `F` is applicable with respect to `A` (Applicable function member).
  - If `F` is generic and `M` has no type argument list, `F` is a candidate when:
    - Type inference (Type inference) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (Satisfying constraints), and the parameter list of `F` is applicable with respect to `A` (Applicable function member).
  - If `F` is generic and `M` includes a type argument list, `F` is a candidate when:
    - `F` has the same number of method type parameters as were supplied in the type argument list, and
    - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (Satisfying constraints), and the parameter list of `F` is applicable with respect to `A` (Applicable function member).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set. Furthermore, if `C` is a class type other than `object`, all methods declared in an interface type are removed from the set. (This latter rule only has affect when the method group was the result of a member lookup on a type parameter having an effective base class other than object and a non-empty effective interface set.)
- If the resulting set of candidate methods is empty, then further processing along the following steps are abandoned, and instead an attempt is made to process the invocation as an extension method invocation

(Extension method invocations). If this fails, then no applicable methods exist, and a binding-time error occurs.

- The best method of the set of candidate methods is identified using the overload resolution rules of Overload resolution. If a single best method cannot be identified, the method invocation is ambiguous, and a binding-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed:
  - The method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a *simple_name* or a *member_access* through a type. If the best method is an instance method, the method group must have resulted from a *simple_name*, a *member_access* through a variable or value, or a *base_access*. If neither of these requirements is true, a binding-time error occurs.
  - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (Satisfying constraints) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a binding-time error occurs.

Once a method has been selected and validated at binding-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in Compile-time checking of dynamic overload resolution.

The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform type inference and overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. If no method was found, try instead to process the invocation as an extension method invocation.

Extension method invocations

In a method invocation (Invocations on boxed instances) of one of the forms

```
expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )
```

if the normal processing of the invocation finds no applicable methods, an attempt is made to process the construct as an extension method invocation. If *expr* or any of the *args* has compile-time type `dynamic`, extension methods will not apply.

The objective is to find the best *type_name* `C`, so that the corresponding static method invocation can take place:

```
C . identifier ( expr )

C . identifier ( expr , args )

C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )
```

An extension method `Ci.Mj` is ***eligible*** if:

- `Ci` is a non-generic, non-nested class
- The name of `Mj` is *identifier*

- `Mj` is accessible and applicable when applied to the arguments as a static method as shown above
- An implicit identity, reference or boxing conversion exists from *expr* to the type of the first parameter of `Mj`.

The search for `C` proceeds as follows:

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
  - If the given namespace or compilation unit directly contains non-generic type declarations `Ci` with eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
  - If types `Ci` imported by *using_static_declarations* and directly declared in namespaces imported by *using_namespace_directive*s in the given namespace or compilation unit directly contain eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
- If no candidate set is found in any enclosing namespace declaration or compilation unit, a compile-time error occurs.
- Otherwise, overload resolution is applied to the candidate set as described in (Overload resolution). If no single best method is found, a compile-time error occurs.
- `C` is the type within which the best method is declared as an extension method.

Using `C` as a target, the method call is then processed as a static method invocation (Compile-time checking of dynamic overload resolution).

The preceding rules mean that instance methods take precedence over extension methods, that extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations, and that extension methods declared directly in a namespace take precedence over extension methods imported into that same namespace with a using namespace directive. For example:

```
public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}

class X
{
    static void Test(A a, B b, C c) {
        a.F(1);             // E.F(object, int)
        a.F("hello");       // E.F(object, string)

        b.F(1);             // B.F(int)
        b.F("hello");       // E.F(object, string)

        c.F(1);             // C.F(object)
        c.F("hello");       // C.F(object)
    }
}
```

In the example, `B`'s method takes precedence over the first extension method, and `C`'s method takes precedence over both extension methods.

```
public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
        public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
    }
}

namespace N2
{
    using N1;

    public static class E
    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i); }
    }

    class Test
    {
        static void Main(string[] args)
        {
            1.F();
            2.G();
            3.H();
        }
    }
}
```

The output of this example is:

```
E.F(1)
D.G(2)
C.H(3)
```

`D.G` takes precedence over `C.G`, and `E.F` takes precedence over both `D.F` and `C.F`.

Delegate invocations

For a delegate invocation, the *primary_expression* of the *invocation_expression* must be a value of a *delegate_type*. Furthermore, considering the *delegate_type* to be a function member with the same parameter list as the *delegate_type*, the *delegate_type* must be applicable (Applicable function member) with respect to the *argument_list* of the *invocation_expression*.

The run-time processing of a delegate invocation of the form `D(A)`, where `D` is a *primary_expression* of a *delegate_type* and `A` is an optional *argument_list*, consists of the following steps:

- `D` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The value of `D` is checked to be valid. If the value of `D` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- Otherwise, `D` is a reference to a delegate instance. Function member invocations (Compile-time checking of dynamic overload resolution) are performed on each of the callable entities in the invocation list of the

delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

Element access

An *element_access* consists of a *primary_no_array_creation_expression*, followed by a "`[`" token, followed by an *argument_list*, followed by a "`]`" token. The *argument_list* consists of one or more *argument*s, separated by commas.

```
element_access
    : primary_no_array_creation_expression '[' expression_list ']'
    ;
```

The *argument_list* of an *element_access* is not allowed to contain `ref` or `out` arguments.

An *element_access* is dynamically bound (Dynamic binding) if at least one of the following holds:

- The *primary_no_array_creation_expression* has compile-time type `dynamic`.
- At least one expression of the *argument_list* has compile-time type `dynamic` and the *primary_no_array_creation_expression* does not have an array type.

In this case the compiler classifies the *element_access* as a value of type `dynamic`. The rules below to determine the meaning of the *element_access* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary_no_array_creation_expression* and *argument_list* expressions which have the compile-time type `dynamic`. If the *primary_no_array_creation_expression* does not have compile-time type `dynamic`, then the element access undergoes a limited compile time check as described in Compile-time checking of dynamic overload resolution.

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 (Array access). 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 (Indexer access).

Array access

For an array access, the *primary_no_array_creation_expression* of the *element_access* must be a value of an *array_type*. Furthermore, the *argument_list* of an array access is not allowed to contain named arguments.The number of expressions in the *argument_list* must be the same as the rank of the *array_type*, and each expression must be of type `int`, `uint`, `long`, `ulong`, or must be implicitly convertible to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value(s) of the expression(s) in the *argument_list*.

The run-time processing of an array access of the form `P[A]`, where `P` is a *primary_no_array_creation_expression* of an *array_type* and `A` is an *argument_list*, consists of the following steps:

- `P` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The index expressions of the *argument_list* are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion (Implicit conversions) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. For instance, if the index expression is of type `short` then an implicit conversion to `int` is performed, since implicit conversions from `short` to `int` and from `short` to `long` are possible. If evaluation of an index expression or the subsequent implicit conversion causes an exception, then no further index expressions are evaluated and no further steps are executed.
- The value of `P` is checked to be valid. If the value of `P` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- The value of each expression in the *argument_list* is checked against the actual bounds of each dimension of

the array instance referenced by `P`. If one or more values are out of range, a `System.IndexOutOfRangeException` is thrown and no further steps are executed.

- The location of the array element given by the index expression(s) is computed, and this location becomes the result of the array access.

Indexer access

For an indexer access, the *primary_no_array_creation_expression* of the *element_access* must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the *argument_list* of the *element_access*.

The binding-time processing of an indexer access of the form `P[A]`, where `P` is a *primary_no_array_creation_expression* of a class, struct, or interface type `T`, and `A` is an *argument_list*, consists of the following steps:

- The set of indexers provided by `T` is constructed. The set consists of all indexers declared in `T` or a base type of `T` that are not `override` declarations and are accessible in the current context (Member access).
- 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` (Applicable function member), then `I` is removed from the set.
  - If `I` is applicable with respect to `A` (Applicable function member), then all indexers declared in a base type of `S` are removed from the set.
  - If `I` is applicable with respect to `A` (Applicable function member) and `S` is a class type other than `object`, all indexers declared in an interface are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a binding-time error occurs.
- The best indexer of the set of candidate indexers is identified using the overload resolution rules of Overload resolution. If a single best indexer cannot be identified, the indexer access is ambiguous, and a binding-time error occurs.
- The index expressions of the *argument_list* are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of `P` and an associated argument list of `A`.

Depending on the context in which it is used, an indexer access causes invocation of either the *get accessor* or the *set accessor* of the indexer. If the indexer access is the target of an assignment, the *set accessor* is invoked to assign a new value (Simple assignment). In all other cases, the *get accessor* is invoked to obtain the current value (Values of expressions).

This access

A *this_access* consists of the reserved word `this`.

```
this_access
    : 'this'
    ;
```

A *this_access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When `this` is used in a *primary_expression* within an instance constructor of a class, it is classified as a value. The type of the value is the instance type (The instance type) of the class within which the usage occurs, and the value is a reference to the object being constructed.
- When `this` is used in a *primary_expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the instance type (The instance type) of the class within which the

usage occurs, and the value is a reference to the object for which the method or accessor was invoked.

- When `this` is used in a *primary_expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the instance type (The instance type) of the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
- When `this` is used in a *primary_expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the instance type (The instance type) of the struct within which the usage occurs.
  - If the method or accessor is not an iterator (Iterators), the `this` variable represents the struct for which the method or accessor was invoked, and behaves exactly the same as a `ref` parameter of the struct type.
  - If the method or accessor is an iterator, the `this` variable represents a copy of the struct for which the method or accessor was invoked, and behaves exactly the same as a value parameter of the struct type.

Use of `this` in a *primary_expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable_initializer* of a field declaration.

Base access

A *base_access* consists of the reserved word `base` followed by either a "`.`" token and an identifier or an *argument_list* enclosed in square brackets:

```
base_access
    : 'base' '.' identifier
    | 'base' '[' 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 binding-time, *base_access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus, `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

When a *base_access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at run-time (Compile-time checking of dynamic overload resolution) is changed. The function member that is invoked is determined by finding the most derived implementation (Virtual methods) of the function member with respect to `B` (instead of with respect to the run-time type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base_access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base_access* is abstract, a binding-time error occurs.

Postfix increment and decrement operators

```
post_increment_expression
    : primary_expression '++'
    ;

post_decrement_expression
    : primary_expression '--'
    ;
```

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the *primary_expression* has the compile-time type `dynamic` then the operator is dynamically bound (Dynamic binding), the *post_increment_expression* or *post_decrement_expression* has the compile-time type `dynamic` and the following rules are applied at run-time using the run-time type of the *primary_expression*.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The run-time processing of a postfix increment or decrement operation of the form `x++` or `x--` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The value of `x` is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The saved value of `x` becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked and the returned value is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
  - The saved value of `x` becomes the result of the operation.

The `++` and `--` operators also support prefix notation (Prefix increment and decrement operators). Typically, the result of `x++` or `x--` is the value of `x` before the operation, whereas the result of `++x` or `--x` is the value of `x` after the operation. In either case, `x` itself has the same value after the operation.

An `operator ++` or `operator --` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

The new operator

The `new` operator is used to create new instances of types.

There are three forms of `new` expressions:

- Object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The `new` operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when `new` is used to create instances of value types.

Object creation expressions

An *object_creation_expression* is used to create a new instance of a *class_type* or a *value_type*.

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    ;

object_or_collection_initializer
    : object_initializer
    | collection_initializer
    ;
```

The *type* of an *object_creation_expression* must be a *class_type*, a *value_type* or a *type_parameter*. The *type* cannot be an `abstract` *class_type*.

The optional *argument_list* (Argument lists) is permitted only if the *type* is a *class_type* or a *struct_type*.

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Processing of an object creation expression that includes an object initializer or collection initializer consists of first processing the instance constructor and then processing the member or element initializations specified by the object initializer (Object initializers) or collection initializer (Collection initializers).

If any of the arguments in the optional *argument_list* has the compile-time type `dynamic` then the *object_creation_expression* is dynamically bound (Dynamic binding) and the following rules are applied at run-time using the run-time type of those arguments of the *argument_list* that have the compile time type `dynamic`. However, the object creation undergoes a limited compile time check as described in Compile-time checking of dynamic overload resolution.

The binding-time processing of an *object_creation_expression* of the form `new T(A)`, where `T` is a *class_type* or a *value_type* and `A` is an optional *argument_list*, consists of the following steps:

- If `T` is a *value_type* and `A` is not present:
  - The *object_creation_expression* is a default constructor invocation. The result of the *object_creation_expression* is a value of type `T`, namely the default value for `T` as defined in The System.ValueType type.
- Otherwise, if `T` is a *type_parameter* and `A` is not present:
  - If no value type constraint or constructor constraint (Type parameter constraints) has been specified for `T`, a binding-time error occurs.
  - The result of the *object_creation_expression* is a value of the run-time type that the type parameter has been bound to, namely the result of invoking the default constructor of that type. The run-time type may be a reference type or a value type.
- Otherwise, if `T` is a *class_type* or a *struct_type*:
  - If `T` is an `abstract` *class_type*, a compile-time error occurs.

    - The instance constructor to invoke is determined using the overload resolution rules of Overload resolution. The set of candidate instance constructors consists of all accessible instance constructors declared in `T` which are applicable with respect to `A` (Applicable function member). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a binding-time error occurs.
    - The result of the *object_creation_expression* is a value of type `T`, namely the value produced by invoking the instance constructor determined in the step above.
- Otherwise, the *object_creation_expression* is invalid, and a binding-time error occurs.

Even if the *object_creation_expression* is dynamically bound, the compile-time type is still `T`.

The run-time processing of an *object_creation_expression* of the form `new T(A)`, where `T` is *class_type* or a *struct_type* and `A` is an optional *argument_list*, consists of the following steps:

- If `T` is a *class_type*:
    - A new instance of class `T` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
    - All fields of the new instance are initialized to their default values (Default values).
    - The instance constructor is invoked according to the rules of function member invocation (Compile-time checking of dynamic overload resolution). 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 (Compile-time checking of dynamic overload resolution). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.

Object initializers

An ***object initializer*** specifies values for zero or more fields, properties or indexed elements of an object.

```
object_initializer
    : '{' member_initializer_list? '}'
    | '{' member_initializer_list ',' '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : initializer_target '=' initializer_value
    ;

initializer_target
    : identifier
    | '[' argument_list ']'
    ;

initializer_value
    : expression
    | object_or_collection_initializer
    ;
```

An object initializer consists of a sequence of member initializers, enclosed by `{` and `}` tokens and separated by commas. Each *member_initializer* designates a target for the initialization. An *identifier* must name an accessible field or property of the object being initialized, whereas an *argument_list* enclosed in square brackets must specify arguments for an accessible indexer on the object being initialized. It is an error for an object initializer to include more than one member initializer for the same field or property.

Each *initializer_target* is followed by an equals sign and either an expression, an object initializer or a collection initializer. It is not possible for expressions within the object initializer to refer to the newly created object it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment (Simple assignment) to the target.

A member initializer that specifies an object initializer after the equals sign is a ***nested object initializer***, i.e. an initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the nested object initializer are treated as assignments to members of the field or property. Nested object initializers cannot be applied to properties with a value type, or to read-only fields with a value type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the target field, property or indexer, the elements given in the initializer are added to the collection referenced by the target. The target must be of a collection type that satisfies the requirements specified in Collection initializers.

The arguments to an index initializer will always be evaluated exactly once. Thus, even if the arguments end up never getting used (e.g. because of an empty nested initializer), they will be evaluated for their side effects.

The following class represents a point with two coordinates:

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

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

An instance of `Point` can be created and initialized as follows:

```
Point a = new Point { X = 0, Y = 1 };
```

which has the same effect as

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

where `__a` is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

An instance of `Rectangle` can be created and initialized as follows:

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

where `__r`, `__p1` and `__p2` are temporary variables that are otherwise invisible and inaccessible.

If `Rectangle`'s constructor allocates the two embedded `Point` instances

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();

    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

the following construct can be used to initialize the embedded `Point` instances instead of assigning new instances:

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

Given an appropriate definition of C, the following example:

```
var c = new C {
    x = true,
    y = { a = "Hello" },
    z = { 1, 2, 3 },
    ["x"] = 5,
    [0,0] = { "a", "b" },
    [1,2] = {}
};
```

is equivalent to this series of assignments:

```
C __c = new C();
__c.x = true;
__c.y.a = "Hello";
__c.z.Add(1);
__c.z.Add(2);
__c.z.Add(3);
string __i1 = "x";
__c[__i1] = 5;
int __i2 = 0, __i3 = 0;
__c[__i2,__i3].Add("a");
__c[__i2,__i3].Add("b");
int __i4 = 1, __i5 = 2;
var c = __c;
```

where `__c`, etc., are generated variables that are invisible and inaccessible to the source code. Note that the arguments for `[0,0]` are evaluated only once, and the arguments for `[1,2]` are evaluated once even though they are never used.

Collection initializers

A collection initializer specifies the elements of a collection.

```
collection_initializer
    : '{' element_initializer_list '}'
    | '{' element_initializer_list ',' '}'
    ;

element_initializer_list
    : element_initializer (',' element_initializer)*
    ;

element_initializer
    : non_assignment_expression
    | '{' expression_list '}'
    ;

expression_list
    : expression (',' expression)*
    ;
```

A collection initializer consists of a sequence of element initializers, enclosed by `{` and `}` tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by `{` and `}` tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment expression, to avoid ambiguity with member initializers. The *non_assignment_expression* production is defined in Expression.

The following is an example of an object creation expression that includes a collection initializer:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

The collection object to which a collection initializer is applied must be of a type that implements `System.Collections.IEnumerable` or a compile-time error occurs. For each specified element in order, the collection initializer invokes an `Add` method on the target object with the expression list of the element initializer as argument list, applying normal member lookup and overload resolution for each invocation. Thus, the collection object must have an applicable instance or extension method with the name `Add` for each element initializer.

The following class represents a contact with a name and a list of phone numbers:

```
public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();

    public string Name { get { return name; } set { name = value; } }

    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

A `List<Contact>` can be created and initialized as follows:

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

which has the same effect as

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

where `__clist`, `__c1` and `__c2` are temporary variables that are otherwise invisible and inaccessible.

Array creation expressions

An *array_creation_expression* is used to create a new instance of an *array_type*.

```
array_creation_expression
    : 'new' non_array_type '[' expression_list ']' rank_specifier* array_initializer?
    | 'new' array_type array_initializer
    | 'new' rank_specifier array_initializer
    ;
```

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression `new int[10,20]` produces an array instance of type `int[,]`, and the array creation expression `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of type `int`, `uint`, `long`, or `ulong`, or implicitly convertible to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension must be nonnegative, it is a compile-time error to have a *constant_expression* with a negative value in the expression list.

Except in an unsafe context (Unsafe contexts), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second or third form, the rank of the specified array type or rank specifier must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

An array creation expression of the third form is referred to as an ***implicitly typed array creation expression***. It is similar to the second form, except that the element type of the array is not explicitly given, but determined as the best common type (Finding the best common type of a set of expressions) of the set of expressions in the array initializer. For a multidimensional array, i.e., one where the *rank_specifier* contains at least one comma, this set comprises all *expression*s found in nested *array_initializer*s.

Array initializers are described further in Array initializers.

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 (Implicit conversions) 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 (Default values).
- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of

such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is `null`. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

```
int[][] a = new int[100][5];        // Error
```

results in a compile-time error. Instantiation of the sub-arrays must instead be performed manually, as in

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

When an array of arrays has a "rectangular" shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects—one outer array and 100 sub-arrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

The following are examples of implicitly typed array creation expressions:

```
var a = new[] { 1, 10, 100, 1000 };                     // int[]

var b = new[] { 1, 1.5, 2, 2.5 };                       // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" };                   // Error
```

The last expression causes a compile-time error because neither `int` nor `string` is implicitly convertible to the other, and so there is no best common type. An explicitly typed array creation expression must be used in this case, for example specifying the type to be `object[]`. Alternatively, one of the elements can be cast to a common base type, which would then become the inferred element type.

Implicitly typed array creation expressions can be combined with anonymous object initializers (Anonymous object creation expressions) to create anonymously typed data structures. For example:

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Delegate creation expressions

A *delegate_creation_expression* is used to create a new instance of a *delegate_type*.

```
delegate_creation_expression
    : 'new' delegate_type '(' expression ')'
    ;
```

The argument of a delegate creation expression must be a method group, an anonymous function or a value of either the compile time type `dynamic` or a *delegate_type*. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is an anonymous function it directly defines the parameters and method body of the delegate target. If the argument is a value it identifies a delegate instance of which to create a copy.

If the *expression* has the compile-time type `dynamic`, the *delegate_creation_expression* is dynamically bound (Dynamic binding), and the rules below are applied at run-time using the run-time type of the *expression*. Otherwise the rules are applied at compile-time.

The binding-time processing of a *delegate_creation_expression* of the form `new D(E)`, where `D` is a *delegate_type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is processed in the same way as a method group conversion (Method group conversions) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation expression is processed in the same way as an anonymous function conversion (Anonymous function conversions) from `E` to `D`.
- If `E` is a value, `E` must be compatible (Delegate declarations) with `D`, and the result is a reference to a newly created delegate of type `D` that refers to the same invocation list as `E`. If `E` is not compatible with `D`, a compile-time error occurs.

The run-time processing of a *delegate_creation_expression* of the form `new D(E)`, where `D` is a *delegate_type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is evaluated as a method group conversion (Method group conversions) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation is evaluated as an anonymous function conversion from `E` to `D` (Anonymous function conversions).
- If `E` is a value of a *delegate_type*:
  - `E` is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - The new delegate instance is initialized with the same invocation list as the delegate instance given by `E`.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When two delegates are combined or one is removed from another (Delegate declarations), a new delegate results; no existing delegate has its contents changed.

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);

class A
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }

    static double Square(double x) {
        return x * x;
    }
}
```

the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method not been present, a compile-time error would have occurred.

Anonymous object creation expressions

An *anonymous_object_creation_expression* is used to create an object of an anonymous type.

```
anonymous_object_creation_expression
    : 'new' anonymous_object_initializer
    ;

anonymous_object_initializer
    : '{' member_declarator_list? '}'
    | '{' member_declarator_list ',' '}'
    ;

member_declarator_list
    : member_declarator (',' member_declarator)*
    ;

member_declarator
    : simple_name
    | member_access
    | base_access
    | null_conditional_member_access
    | identifier '=' expression
    ;
```

An anonymous object initializer declares an anonymous type and returns an instance of that type. An anonymous type is a nameless class type that inherits directly from `object`. The members of an anonymous type are a sequence of read-only properties inferred from the anonymous object initializer used to create an instance of the type. Specifically, an anonymous object initializer of the form

```
new { p1 = e1, p2 = e2, ..., pn = en }
```

declares an anonymous type of the form

```
class __Anonymous1
{
    private readonly T1 f1;
    private readonly T2 f2;
    ...
    private readonly Tn fn;

    public __Anonymous1(T1 a1, T2 a2, ..., Tn an) {
        f1 = a1;
        f2 = a2;
        ...
        fn = an;
    }

    public T1 p1 { get { return f1; } }
    public T2 p2 { get { return f2; } }
    ...
    public Tn pn { get { return fn; } }

    public override bool Equals(object __o) { ... }
    public override int GetHashCode() { ... }
}
```

where each `Tx` is the type of the corresponding expression `ex`. The expression used in a *member_declarator* must have a type. Thus, it is a compile-time error for an expression in a *member_declarator* to be null or an anonymous function. It is also a compile-time error for the expression to have an unsafe type.

The names of an anonymous type and of the parameter to its `Equals` method are automatically generated by the compiler and cannot be referenced in program text.

Within the same program, two anonymous object initializers that specify a sequence of properties of the same names and compile-time types in the same order will produce instances of the same anonymous type.

In the example

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

the assignment on the last line is permitted because `p1` and `p2` are of the same anonymous type.

The `Equals` and `GetHashcode` methods on anonymous types override the methods inherited from `object`, and are defined in terms of the `Equals` and `GetHashcode` of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name (Type inference), a member access (Compile-time checking of dynamic overload resolution), a base access (Base access) or a null-conditional member access (Null-conditional expressions as projection initializers). This is called a ***projection initializer*** and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

```
identifier
expr.identifier
```

are precisely equivalent to the following, respectively:

```
identifier = identifier
identifier = expr.identifier
```

Thus, in a projection initializer the *identifier* selects both the value and the field or property to which the value is assigned. Intuitively, a projection initializer projects not just a value, but also the name of the value.

The typeof operator

The `typeof` operator is used to obtain the `System.Type` object for a type.

```
typeof_expression
    : 'typeof' '(' type ')'
    | 'typeof' '(' unbound_type_name ')'
    | 'typeof' '(' 'void' ')'
    ;

unbound_type_name
    : identifier generic_dimension_specifier?
    | identifier '::' identifier generic_dimension_specifier?
    | unbound_type_name '.' identifier generic_dimension_specifier?
    ;

generic_dimension_specifier
    : '<' comma* '>'
    ;

comma
    : ','
    ;
```

The first form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized *type*. The result of an expression of this form is the `System.Type` object for the indicated type. There is only one `System.Type` object for any given type. This means that for a type `T`, `typeof(T) == typeof(T)` is always true. The *type* cannot be `dynamic`.

The second form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized *unbound_type_name*. An *unbound_type_name* is very similar to a *type_name* (Namespace and type names) except that an *unbound_type_name* contains *generic_dimension_specifier*s where a *type_name* contains *type_argument_list*s. When the operand of a *typeof_expression* is a sequence of tokens that satisfies the grammars of both *unbound_type_name* and *type_name*, namely when it contains neither a *generic_dimension_specifier* nor a *type_argument_list*, the sequence of tokens is considered to be a *type_name*. The meaning of an *unbound_type_name* is determined as follows:

- Convert the sequence of tokens to a *type_name* by replacing each *generic_dimension_specifier* with a *type_argument_list* having the same number of commas and the keyword `object` as each *type_argument*.
- Evaluate the resulting *type_name*, while ignoring all type parameter constraints.
- The *unbound_type_name* resolves to the unbound generic type associated with the resulting constructed type (Bound and unbound types).

The result of the *typeof_expression* is the `System.Type` object for the resulting unbound generic type.

The third form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized `void` keyword. The result of an expression of this form is the `System.Type` object that represents the absence of a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type. This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of `System.Type`.

The `typeof` operator can be used on a type parameter. The result is the `System.Type` object for the run-time type that was bound to the type parameter. The `typeof` operator can also be used on a constructed type or an unbound generic type (Bound and unbound types). The `System.Type` object for an unbound generic type is not the same as the `System.Type` object of the instance type. The instance type is always a closed constructed type at run-time so

its `System.Type` object depends on the run-time type arguments in use, while the unbound generic type has no type arguments.

The example

```
using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}

class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

produces the following output:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

Note that `int` and `System.Int32` are the same type.

Also note that the result of `typeof(X<>)` does not depend on the type argument but the result of `typeof(X<T>)` does.

The checked and unchecked operators

The `checked` and `unchecked` operators are used to control the ***overflow checking context*** for integral-type arithmetic operations and conversions.

```
checked_expression
    : 'checked' '(' expression ')'
    ;

unchecked_expression
    : 'unchecked' '(' expression ')'
    ;
```

The `checked` operator evaluates the contained expression in a checked context, and the `unchecked` operator evaluates the contained expression in an unchecked context. A *checked_expression* or *unchecked_expression* corresponds exactly to a *parenthesized_expression* (Parenthesized expressions), 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 (The checked and unchecked statements).

The following operations are affected by the overflow checking context established by the `checked` and `unchecked` operators and statements:

- The predefined `++` and `--` unary operators (Postfix increment and decrement operators and Prefix increment and decrement operators), when the operand is of an integral type.
- The predefined `-` unary operator (Unary minus operator), when the operand is of an integral type.
- The predefined `+`, `-`, `*`, and `/` binary operators (Arithmetic operators), when both operands are of integral types.
- Explicit numeric conversions (Explicit numeric conversions) from one integral type to another integral type, or from `float` or `double` to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a `checked` context, if the operation is a constant expression (Constant expressions), a compile-time error occurs. Otherwise, when the operation is performed at run-time, a `System.OverflowException` is thrown.
- In an `unchecked` context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For non-constant expressions (expressions that are evaluated at run-time) that are not enclosed by any `checked` or `unchecked` operators or statements, the default overflow checking context is `unchecked` unless external factors (such as compiler switches and execution environment configuration) call for `checked` evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always `checked`. Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

The body of an anonymous function is not affected by `checked` or `unchecked` contexts in which the anonymous function occurs.

In the example

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);      // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Depends on default
    }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the `F` method throws a `System.OverflowException`, and the `G` method returns -727379968 (the lower 32 bits of the out-of-range result). The behavior of the `H` method depends on the default overflow checking context for the compilation, but it is either the same as `F` or the same as `G`.

In the example

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);      // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Compile error, overflow
    }
}
```

the overflows that occur when evaluating the constant expressions in `F` and `H` cause compile-time errors to be reported because the expressions are evaluated in a `checked` context. An overflow also occurs when evaluating the constant expression in `G`, but since the evaluation takes place in an `unchecked` context, the overflow is not reported.

The `checked` and `unchecked` operators only affect the overflow checking context for those operations that are textually contained within the "`(`" and "`)`" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. In the example

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

the use of `checked` in `F` does not affect the evaluation of `x * y` in `Multiply`, so `x * y` is evaluated in the default overflow checking context.

The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type `uint`. Because the constants are outside the `int` range, without the `unchecked` operator, the casts to `int` would produce compile-time errors.

The `checked` and `unchecked` operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow even within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an exception on overflow even within an explicitly `checked` construct. In addition, other operators are never affected by the mode of checking, whether default or explicit.

Default value expressions

A default value expression is used to obtain the default value (Default values) of a type. Typically a default value expression is used for type parameters, since it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the `null` literal to a type parameter unless the type parameter is known to be a reference type.)

```
default_value_expression
    : 'default' '(' type ')'
    ;
```

If the *type* in a *default_value_expression* evaluates at run-time to a reference type, the result is `null` converted to that type. If the *type* in a *default_value_expression* evaluates at run-time to a value type, the result is the *value_type*'s default value (Default constructors).

A *default_value_expression* is a constant expression (Constant expressions) if the type is a reference type or a type parameter that is known to be a reference type (Type parameter constraints). In addition, a *default_value_expression* is a constant expression if the type is one of the following value types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, or any enumeration type.

Nameof expressions

A *nameof_expression* is used to obtain the name of a program entity as a constant string.

```
nameof_expression
    : 'nameof' '(' named_entity ')'
    ;

named_entity
    : simple_name
    | named_entity_target '.' identifier type_argument_list?
    ;

named_entity_target
    : 'this'
    | 'base'
    | named_entity
    | predefined_type
    | qualified_alias_member
    ;
```

Grammatically speaking, the *named_entity* operand is always an expression. Because `nameof` is not a reserved keyword, a nameof expression is always syntactically ambiguous with an invocation of the simple name `nameof`. For compatibility reasons, if a name lookup (Simple names) of the name `nameof` succeeds, the expression is treated as an *invocation_expression* -- regardless of whether the invocation is legal. Otherwise it is a *nameof_expression*.

The meaning of the *named_entity* of a *nameof_expression* is the meaning of it as an expression; that is, either as a *simple_name*, a *base_access* or a *member_access*. However, where the lookup described in Simple names and Member access results in an error because an instance member was found in a static context, a *nameof_expression* produces no such error.

It is a compile-time error for a *named_entity* designating a method group to have a *type_argument_list*. It is a compile time error for a *named_entity_target* to have the type `dynamic`.

A *nameof_expression* is a constant expression of type `string`, and has no effect at runtime. Specifically, its *named_entity* is not evaluated, and is ignored for the purposes of definite assignment analysis (General rules for simple expressions). Its value is the last identifier of the *named_entity* before the optional final *type_argument_list*, transformed in the following way:

- The prefix "`@`", if used, is removed.
- Each *unicode_escape_sequence* is transformed into its corresponding Unicode character.
- Any *formatting_characters* are removed.

These are the same transformations applied in Identifiers when testing equality between identifiers.

TODO: examples

Anonymous method expressions

An *anonymous_method_expression* is one of two ways of defining an anonymous function. These are further described in Anonymous function expressions.

Unary operators

The `?`, `+`, `-`, `!`, `~`, `++`, `--`, cast, and `await` operators are called the unary operators.

```
unary_expression
    : primary_expression
    | null_conditional_expression
    | '+' unary_expression
    | '-' unary_expression
    | '!' unary_expression
    | '~' unary_expression
    | pre_increment_expression
    | pre_decrement_expression
    | cast_expression
    | await_expression
    | unary_expression_unsafe
    ;
```

If the operand of a *unary_expression* has the compile-time type `dynamic`, it is dynamically bound (Dynamic binding). In this case the compile-time type of the *unary_expression* is `dynamic`, and the resolution described below will take place at run-time using the run-time type of the operand.

Null-conditional operator

The null-conditional operator applies a list of operations to its operand only if that operand is non-null. Otherwise the result of applying the operator is `null`.

```
null_conditional_expression
    : primary_expression null_conditional_operations
    ;

null_conditional_operations
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;
```

The list of operations can include member access and element access operations (which may themselves be null-conditional), as well as invocation.

For example, the expression `a.b?[0]?.c()` is a *null_conditional_expression* with a *primary_expression* `a.b` and *null_conditional_operations* `?[0]` (null-conditional element access), `?.c` (null-conditional member access) and `()` (invocation).

For a *null_conditional_expression* `E` with a *primary_expression* `P`, let `E0` be the expression obtained by textually removing the leading `?` from each of the *null_conditional_operations* of `E` that have one. Conceptually, `E0` is the expression that will be evaluated if none of the null checks represented by the `?`s do find a `null`.

Also, let `E1` be the expression obtained by textually removing the leading `?` from just the first of the *null_conditional_operations* in `E`. This may lead to a *primary-expression* (if there was just one `?`) or to another *null_conditional_expression*.

For example, if `E` is the expression `a.b?[0]?.c()`, then `E0` is the expression `a.b[0].c()` and `E1` is the expression `a.b[0]?.c()`.

If `E0` is classified as nothing, then `E` is classified as nothing. Otherwise E is classified as a value.

`E0` and `E1` are used to determine the meaning of `E`:

- If `E` occurs as a *statement_expression* the meaning of `E` is the same as the statement

```
if ((object)P != null) E1;
```

except that P is evaluated only once.

- Otherwise, if `E0` is classified as nothing a compile-time error occurs.
- Otherwise, let `T0` be the type of `E0`.
  - If `T0` is a type parameter that is not known to be a reference type or a non-nullable value type, a compile-time error occurs.
  - If `T0` is a non-nullable value type, then the type of `E` is `T0?`, and the meaning of `E` is the same as

    ```
    ((object)P == null) ? (T0?)null : E1
    ```

    except that `P` is evaluated only once.
  - Otherwise the type of E is T0, and the meaning of E is the same as

    ```
    ((object)P == null) ? null : E1
    ```

    except that `P` is evaluated only once.

If `E1` is itself a *null_conditional_expression*, then these rules are applied again, nesting the tests for `null` until there are no further `?`'s, and the expression has been reduced all the way down to the primary-expression `E0`.

For example, if the expression `a.b?[0]?.c()` occurs as a statement-expression, as in the statement:

```
a.b?[0]?.c();
```

its meaning is equivalent to:

```
if (a.b != null) a.b[0]?.c();
```

which again is equivalent to:

```
if (a.b != null) if (a.b[0] != null) a.b[0].c();
```

Except that `a.b` and `a.b[0]` are evaluated only once.

If it occurs in a context where its value is used, as in:

```
var x = a.b?[0]?.c();
```

and assuming that the type of the final invocation is not a non-nullable value type, its meaning is equivalent to:

```
var x = (a.b == null) ? null : (a.b[0] == null) ? null : a.b[0].c();
```

except that `a.b` and `a.b[0]` are evaluated only once.

Null-conditional expressions as projection initializers

A null-conditional expression is only allowed as a *member_declarator* in an

anonymous_object_creation_expression (Anonymous object creation expressions) if it ends with an (optionally null-conditional) member access. Grammatically, this requirement can be expressed as:

```
null_conditional_member_access
    : primary_expression null_conditional_operations? '?' '.' identifier type_argument_list?
    | primary_expression null_conditional_operations '.' identifier type_argument_list?
    ;
```

This is a special case of the grammar for *null_conditional_expression* above. The production for *member_declarator* in Anonymous object creation expressions then includes only *null_conditional_member_access*.

Null-conditional expressions as statement expressions

A null-conditional expression is only allowed as a *statement_expression* (Expression statements) if it ends with an invocation. Grammatically, this requirement can be expressed as:

```
null_conditional_invocation_expression
    : primary_expression null_conditional_operations '(' argument_list? ')'
    ;
```

This is a special case of the grammar for *null_conditional_expression* above. The production for *statement_expression* in Expression statements then includes only *null_conditional_invocation_expression*.

Unary plus operator

For an operation of the form `+x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

For each of these operators, the result is simply the value of the operand.

Unary minus operator

For an operation of the form `-x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined negation operators are:

- Integer negation:

  ```
  int operator -(int x);
  long operator -(long x);
  ```

  The result is computed by subtracting `x` from zero. If the value of of `x` is the smallest representable value of the operand type (-2^31 for `int` or -2^63 for `long`), then the mathematical negation of `x` is not representable within the operand type. If this occurs within a `checked` context, a `System.OverflowException` is thrown; if it occurs within an `unchecked` context, the result is the value of the operand and the overflow is

not reported.

If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the result is `long`. An exception is the rule that permits the `int` value -2147483648 (-2^31) to be written as a decimal integer literal (Integer literals).

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 (Integer literals).

- Floating-point negation:

```
float operator -(float x);
double operator -(double x);
```

The result is the value of `x` with its sign inverted. If `x` is NaN, the result is also NaN.

- Decimal negation:

```
decimal operator -(decimal x);
```

The result is computed by subtracting `x` from zero. Decimal negation is equivalent to using the unary minus operator of type `System.Decimal`.

Logical negation operator

For an operation of the form `!x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

```
bool operator !(bool x);
```

This operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`.

Bitwise complement operator

For an operation of the form `~x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

For each of these operators, the result of the operation is the bitwise complement of `x`.

Every enumeration type `E` implicitly provides the following bitwise complement operator:

```
E operator ~(E x);
```

The result of evaluating `~x`, where `x` is an expression of an enumeration type `E` with an underlying type `U`, is

exactly the same as evaluating `(E)(~(U)x)`, except that the conversion to `E` is always performed as if in an `unchecked` context (The checked and unchecked operators).

Prefix increment and decrement operators

```
pre_increment_expression
    : '++' unary_expression
    ;

pre_decrement_expression
    : '--' unary_expression
    ;
```

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The run-time processing of a prefix increment or decrement operation of the form `++x` or `--x` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The selected operator is invoked with the value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The value returned by the operator becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked.
  - The selected operator is invoked with the value returned by the `get` accessor as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
  - The value returned by the operator becomes the result of the operation.

The `++` and `--` operators also support postfix notation (Postfix increment and decrement operators). Typically, the result of `x++` or `x--` is the value of `x` before the operation, whereas the result of `++x` or `--x` is the value of `x` after the operation. In either case, `x` itself has the same value after the operation.

An `operator++` or `operator--` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

Cast expressions

A *cast_expression* is used to explicitly convert an expression to a given type.

```
cast_expression
    : '(' type ')' unary_expression
    ;
```

A *cast_expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary_expression*, performs an explicit conversion (Explicit conversions) of the value of `E` to type `T`. If no explicit conversion exists from `E` to `T`, a binding-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if `E` denotes a variable.

The grammar for a *cast_expression* leads to certain syntactic ambiguities. For example, the expression `(x)-y` could either be interpreted as a *cast_expression* (a cast of `-y` to type `x`) or as an *additive_expression* combined with a *parenthesized_expression* (which computes the value `x - y)`.

To resolve *cast_expression* ambiguities, the following rule exists: A sequence of one or more *token*s (White space) 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* (Unicode character escape sequences), a *literal* (Literals), or any *keyword* (Keywords) except `as` and `is`.

The term "correct grammar" above means only that the sequence of tokens must conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. For example, if `x` and `y` are identifiers, then `x.y` is correct grammar for a type, even if `x.y` doesn't actually denote a type.

From the disambiguation rule it follows that, if `x` and `y` are identifiers, `(x)y`, `(x)(y)`, and `(x)(-y)` are *cast_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).

Await expressions

The await operator is used to suspend evaluation of the enclosing async function until the asynchronous operation represented by the operand has completed.

```
await_expression
    : 'await' unary_expression
    ;
```

An *await_expression* is only allowed in the body of an async function (Iterators). Within the nearest enclosing async function, an *await_expression* may not occur in these places:

- Inside a nested (non-async) anonymous function
- Inside the block of a *lock_statement*
- In an unsafe context

Note that an *await_expression* cannot occur in most places within a *query_expression*, because those are syntactically transformed to use non-async lambda expressions.

Inside of an async function, `await` cannot be used as an identifier. There is therefore no syntactic ambiguity between await-expressions and various expressions involving identifiers. Outside of async functions, `await` acts as a normal identifier.

The operand of an *await_expression* is called the ***task***. It represents an asynchronous operation that may or may not be complete at the time the *await_expression* is evaluated. The purpose of the await operator is to suspend

execution of the enclosing async function until the awaited task is complete, and then obtain its outcome.

Awaitable expressions

The task of an await expression is required to be ***awaitable***. An expression `t` is awaitable if one of the following holds:

- `t` is of compile time type `dynamic`
- `t` has an accessible instance or extension method called `GetAwaiter` with no parameters and no type parameters, and a return type `A` for which all of the following hold:
  - `A` implements the interface `System.Runtime.CompilerServices.INotifyCompletion` (hereafter known as `INotifyCompletion` for brevity)
  - `A` has an accessible, readable instance property `IsCompleted` of type `bool`
  - `A` has an accessible instance method `GetResult` with no parameters and no type parameters

The purpose of the `GetAwaiter` method is to obtain an ***awaiter*** for the task. The type `A` is called the ***awaiter type*** for the await expression.

The purpose of the `IsCompleted` property is to determine if the task is already complete. If so, there is no need to suspend evaluation.

The purpose of the `INotifyCompletion.OnCompleted` method is to sign up a "continuation" to the task; i.e. a delegate (of type `System.Action`) that will be invoked once the task is complete.

The purpose of the `GetResult` method is to obtain the outcome of the task once it is complete. This outcome may be successful completion, possibly with a result value, or it may be an exception which is thrown by the `GetResult` method.

Classification of await expressions

The expression `await t` is classified the same way as the expression `(t).GetAwaiter().GetResult()`. Thus, if the return type of `GetResult` is `void`, the *await_expression* is classified as nothing. If it has a non-void return type `T`, the *await_expression* is classified as a value of type `T`.

Runtime evaluation of await expressions

At runtime, the expression `await t` is evaluated as follows:

- An awaiter `a` is obtained by evaluating the expression `(t).GetAwaiter()`.
- A `bool` `b` is obtained by evaluating the expression `(a).IsCompleted`.
- If `b` is `false` then evaluation depends on whether `a` implements the interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion` (hereafter known as `ICriticalNotifyCompletion` for brevity). This check is done at binding time; i.e. at runtime if `a` has the compile time type `dynamic`, and at compile time otherwise. Let `r` denote the resumption delegate (Iterators):
  - If `a` does not implement `ICriticalNotifyCompletion`, then the expression `(a as (INotifyCompletion)).OnCompleted(r)` is evaluated.
  - If `a` does implement `ICriticalNotifyCompletion`, then the expression `(a as (ICriticalNotifyCompletion)).UnsafeOnCompleted(r)` is evaluated.
  - Evaluation is then suspended, and control is returned to the current caller of the async function.
- Either immediately after (if `b` was `true`), or upon later invocation of the resumption delegate (if `b` was `false`), the expression `(a).GetResult()` is evaluated. If it returns a value, that value is the result of the *await_expression*. Otherwise the result is nothing.

An awaiter's implementation of the interface methods `INotifyCompletion.OnCompleted` and `ICriticalNotifyCompletion.UnsafeOnCompleted` should cause the delegate `r` to be invoked at most once. Otherwise, the behavior of the enclosing async function is undefined.

Arithmetic operators

The `*`, `/`, `%`, `+`, and `-` operators are called the arithmetic operators.

```
multiplicative_expression
    : unary_expression
    | multiplicative_expression '*' unary_expression
    | multiplicative_expression '/' unary_expression
    | multiplicative_expression '%' unary_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
```

If an operand of an arithmetic operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

Multiplication operator

For an operation of the form `x * y`, binary operator overload resolution (Binary operator overload resolution) 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	+inf	-inf	NaN
+x	+z	-z	+0	-0	+inf	-inf	NaN

-x	-z	+z	-0	+0	-inf	+inf	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+inf	+inf	-inf	NaN	NaN	+inf	-inf	NaN
-inf	-inf	+inf	NaN	NaN	-inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

Division operator

For an operation of the form `x / y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of `x` and `y`.

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

The division rounds the result towards zero. Thus the absolute value of the result is the largest possible integer that is less than or equal to the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have opposite signs.

If the left operand is the smallest representable `int` or `long` value and the right operand is `-1`, an overflow occurs. In a `checked` context, this causes a `System.ArithmeticException` (or a subclass thereof) to be thrown. In an `unchecked` context, it is implementation-defined as to whether a `System.ArithmeticException` (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand.

- Floating-point division:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

The quotient is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-z	+inf	-inf	+0	-0	NaN
-x	-z	+z	-inf	+inf	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+inf	+inf	-inf	+inf	-inf	NaN	NaN	NaN
-inf	-inf	+inf	-inf	+inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal division:

```
decimal operator /(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representable decimal value to the true mathematical result.

Decimal division is equivalent to using the division operator of type `System.Decimal`.

Remainder operator

For an operation of the form `x % y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between `x` and `y`.

- Integer remainder:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

The result of `x % y` is the value produced by `x - (x / y) * y`. If `y` is zero, a

`System.DivideByZeroException` is thrown.

If the left operand is the smallest `int` or `long` value and the right operand is `-1`, a `System.OverflowException` is thrown. In no case does `x % y` throw an exception where `x / y` would not throw an exception.

- Floating-point remainder:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x % y` and is computed as `x - n * y`, where `n` is the largest possible integer that is less than or equal to `x / y`. This method of computing the remainder is analogous to that used for integer operands, but differs from the IEEE 754 definition (in which `n` is the integer closest to `x / y`).

	+y	-y	+0	-0	+inf	-inf	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
+inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal remainder:

```
decimal operator %(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of `x`.

Decimal remainder is equivalent to using the remainder operator of type `System.Decimal`.

Addition operator

For an operation of the form `x + y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or both operands are of type string, the predefined addition operators concatenate the string representation of the operands.

- Integer addition:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

In a `checked` context, if the sum is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point addition:

```
float operator +(float x, float y);
double operator +(double x, double y);
```

The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x + y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x + y`.

	y	+0	-0	+inf	-inf	NaN
x	z	x	x	+inf	-inf	NaN
+0	y	+0	+0	+inf	-inf	NaN
-0	y	+0	-0	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal addition is equivalent to using the addition operator of type `System.Decimal`.

- Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where `E` is the enum type, and `U` is the underlying type of `E`:

```
E operator +(E x, U y);
E operator +(U x, E y);
```

At run-time these operators are evaluated exactly as `(E)((U)x + (U)y)`.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

These overloads of the binary `+` operator perform string concatenation. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted.

```
using System;

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");        // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);               // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);               // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);               // displays d = 2.900
    }
}
```

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

- Delegate combination. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

  ```
  D operator +(D x, D y);
  ```

  The binary `+` operator performs delegate combination when both operands are of some delegate type `D`. (If the operands have different delegate types, a binding-time error occurs.) If the first operand is `null`, the result of the operation is the value of the second operand (even if that is also `null`). Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see Subtraction operator and Delegate invocation. Since `System.Delegate` is not a delegate type, `operator +` is not defined for it.

Subtraction operator

For an operation of the form `x - y`, binary operator overload resolution (Binary operator overload resolution) 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`.

NaN	y	+0	-0	+inf	-inf	NaN
x	z	x	x	-inf	+inf	NaN
+0	-y	+0	+0	-inf	+inf	NaN
-0	-y	-0	+0	-inf	+inf	NaN
+inf	+inf	+inf	+inf	NaN	+inf	NaN
-inf	-inf	-inf	-inf	-inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal subtraction:

```
decimal operator -(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type `System.Decimal`.

- Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where `E` is the enum type, and `U` is the underlying type of `E`:

```
U operator -(E x, E y);
```

This operator is evaluated exactly as `(U)((U)x - (U)y)`. In other words, the operator computes the difference between the ordinal values of `x` and `y`, and the type of the result is the underlying type of the enumeration.

```
E operator -(E x, U y);
```

This operator is evaluated exactly as `(E)((U)x - y)`. In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

- Delegate removal. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator -(D x, D y);
```

The binary `-` operator performs delegate removal when both operands are of some delegate type `D`. If the operands have different delegate types, a binding-time error occurs. If the first operand is `null`, the result of the operation is `null`. Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists (Delegate declarations) having one or more entries, and the result is a new invocation list consisting of the first operand's list with the second operand's entries removed from it, provided the second operand's list is a proper contiguous sublist of the first's. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator (Delegate equality operators).) Otherwise, the result is the value of the left operand. Neither of the operands' lists is changed in the process. If the second operand's list matches multiple sublists of contiguous entries in the first operand's list, the right-most matching sublist of contiguous entries is removed. If removal results in an empty list, the result is `null`. For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1;   // M1 + M2 + M2 + M1
        cd3 -= cd1;                      // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;                // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;                // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;                // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;                // => M1 + M2 + M2 + M1
    }
}
```

Shift operators

The `<<` and `>>` operators are used to perform bit shifting operations.

```
shift_expression
    : additive_expression
    | shift_expression '<<' additive_expression
    | shift_expression right_shift additive_expression
    ;
```

If an operand of a *shift_expression* has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x << count` or `x >> count`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be `int`.

The predefined shift operators are listed below.

- Shift left:

  ```
  int operator <<(int x, int count);
  uint operator <<(uint x, int count);
  long operator <<(long x, int count);
  ulong operator <<(ulong x, int count);
  ```

  The `<<` operator shifts `x` left by a number of bits computed as described below.

  The high-order bits outside the range of the result type of `x` are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

- Shift right:

  ```
  int operator >>(int x, int count);
  uint operator >>(uint x, int count);
  long operator >>(long x, int count);
  ulong operator >>(ulong x, int count);
  ```

  The `>>` operator shifts `x` right by a number of bits computed as described below.

  When `x` is of type `int` or `long`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if `x` is non-negative and set to one if `x` is negative.

  When `x` is of type `uint` or `ulong`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

For the predefined operators, the number of bits to shift is computed as follows:

- When the type of `x` is `int` or `uint`, the shift count is given by the low-order five bits of `count`. In other words, the shift count is computed from `count & 0x1F`.
- When the type of `x` is `long` or `ulong`, the shift count is given by the low-order six bits of `count`. In other words, the shift count is computed from `count & 0x3F`.

If the resulting shift count is zero, the shift operators simply return the value of `x`.

Shift operations never cause overflows and produce the same results in `checked` and `unchecked` contexts.

When the left operand of the `>>` operator is of a signed integral type, the operator performs an arithmetic shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the `>>` operator is of an unsigned integral type, the operator performs a logical shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. For example, if `x` is a variable of type `int`, the operation `unchecked((int)((uint)x >> y))` performs a logical shift right of `x`.

Relational and type-testing operators

The `==`, `!=`, `<`, `>`, `<=`, `>=`, `is` and `as` operators are called the relational and type-testing operators.

```
relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression '<=' shift_expression
    | relational_expression '>=' shift_expression
    | relational_expression 'is' type
    | relational_expression 'as' type
    ;

equality_expression
    : relational_expression
    | equality_expression '==' relational_expression
    | equality_expression '!=' relational_expression
    ;
```

The `is` operator is described in The is operator and the `as` operator is described in The as operator.

The `==`, `!=`, `<`, `>`, `<=` and `>=` operators are ***comparison operators***.

If an operand of a comparison operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x` *op* `y`, where *op* is a comparison operator, overload resolution (Binary operator overload resolution) 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

OPERATION	RESULT
`x >= y`	`true` if `x` is greater than or equal to `y`, `false` otherwise

Integer comparison operators

The predefined integer comparison operators are:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Each of these operators compares the numeric values of the two integer operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`.

Floating-point comparison operators

The predefined floating-point comparison operators are:

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

The operators compare the operands according to the rules of the IEEE 754 standard:

- If either operand is NaN, the result is `false` for all operators except `!=`, for which the result is `true`. For any two operands, `x != y` always produces the same result as `!(x == y)`. However, when one or both operands are NaN, the `<`, `>`, `<=`, and `>=` operators do not produce the same results as the logical negation of the opposite operator. For example, if either of `x` and `y` is NaN, then `x < y` is `false`, but `!(x >= y)` is `true`.
- When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

  ```
  -inf < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +inf
  ```

  where `min` and `max` are the smallest and largest positive finite values that can be represented in the given floating-point format. Notable effects of this ordering are:

  - Negative and positive zeros are considered equal.
  - A negative infinity is considered less than all other values, but equal to another negative infinity.
  - A positive infinity is considered greater than all other values, but equal to another positive infinity.

Decimal comparison operators

The predefined decimal comparison operators are:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Each of these operators compares the numeric values of the two decimal operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type `System.Decimal`.

Boolean equality operators

The predefined boolean equality operators are:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

The result of `==` is `true` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `false`.

The result of `!=` is `false` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `true`. When the operands are of type `bool`, the `!=` operator produces the same result as the `^` operator.

Enumeration comparison operators

Every enumeration type implicitly provides the following predefined comparison operators:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and `op` is one of the comparison operators, is exactly the same as evaluating `((U)x) op ((U)y)`. In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

Reference type equality operators

The predefined reference type equality operators are:

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

The operators return the result of comparing the two references for equality or non-equality.

Since the predefined reference type equality operators accept operands of type `object`, they apply to all types that do not declare applicable `operator ==` and `operator !=` members. Conversely, any applicable user-defined equality operators effectively hide the predefined reference type equality operators.

The predefined reference type equality operators require one of the following:

- Both operands are a value of a type known to be a *reference_type* or the literal `null`. Furthermore, an explicit reference conversion (Explicit reference conversions) exists from the type of either operand to the type of the other operand.
- One operand is a value of type `T` where `T` is a *type_parameter* and the other operand is the literal `null`. Furthermore `T` does not have the value type constraint.

Unless one of these conditions are true, a binding-time error occurs. Notable implications of these rules are:

- It is a binding-time error to use the predefined reference type equality operators to compare two references that are known to be different at binding-time. For example, if the binding-time types of the operands are two class types `A` and `B`, and if neither `A` nor `B` derives from the other, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a binding-time error.
- The predefined reference type equality operators do not permit value type operands to be compared. Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of that struct type.
- The predefined reference type equality operators never cause boxing operations to occur for their operands. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.
- If an operand of a type parameter type `T` is compared to `null`, and the run-time type of `T` is a value type, the result of the comparison is `false`.

The following example checks whether an argument of an unconstrained type parameter type is `null`.

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

The `x == null` construct is permitted even though `T` could represent a value type, and the result is simply defined to be `false` when `T` is a value type.

For an operation of the form `x == y` or `x != y`, if any applicable `operator ==` or `operator !=` exists, the operator overload resolution (Binary operator overload resolution) rules will select that operator instead of the predefined

reference type equality operator. However, it is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type `object`. The example

```
using System;

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

produces the output

```
True
False
False
False
```

The `s` and `t` variables refer to two distinct `string` instances containing the same characters. The first comparison outputs `True` because the predefined string equality operator (String equality operators) is selected when both operands are of type `string`. The remaining comparisons all output `False` because the predefined reference type equality operator is selected when one or both of the operands are of type `object`.

Note that the above technique is not meaningful for value types. The example

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

outputs `False` because the casts create references to two separate instances of boxed `int` values.

String equality operators

The predefined string equality operators are:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two `string` values are considered equal when one of the following is true:

- Both values are `null`.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string values rather than string references. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. As described in Reference type equality operators, the reference type equality operators can be used to

compare string references instead of string values.

Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Two delegate instances are considered equal as follows:

- If either of the delegate instances is `null`, they are equal if and only if both are `null`.
- If the delegates have different run-time type they are never equal.
- If both of the delegate instances have an invocation list (Delegate declarations), those instances are equal if and only if their invocation lists are the same length, and each entry in one's invocation list is equal (as defined below) to the corresponding entry, in order, in the other's invocation list.

The following rules govern the equality of invocation list entries:

- If two invocation list entries both refer to the same static method then the entries are equal.
- If two invocation list entries both refer to the same non-static method on the same target object (as defined by the reference equality operators) then the entries are equal.
- Invocation list entries produced from evaluation of semantically identical *anonymous_method_expression*s or *lambda_expression*s with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.

Equality operators and null

The `==` and `!=` operators permit one operand to be a value of a nullable type and the other to be the `null` literal, even if no predefined or user-defined operator (in unlifted or lifted form) exists for the operation.

For an operation of one of the forms

```
x == null
null == x
x != null
null != x
```

where `x` is an expression of a nullable type, if operator overload resolution (Binary operator overload resolution) fails to find an applicable operator, the result is instead computed from the `HasValue` property of `x`. Specifically, the first two forms are translated into `!x.HasValue`, and last two forms are translated into `x.HasValue`.

The is operator

The `is` operator is used to dynamically check if the run-time type of an object is compatible with a given type. The result of the operation `E is T`, where `E` is an expression and `T` is a type, is a boolean value indicating whether `E` can successfully be converted to type `T` by a reference conversion, a boxing conversion, or an unboxing conversion. The operation is evaluated as follows, after type arguments have been substituted for all type parameters:

- If `E` is an anonymous function, a compile-time error occurs
- If `E` is a method group or the `null` literal, of if the type of `E` is a reference type or a nullable type and the value of `E` is null, the result is false.
- Otherwise, let `D` represent the dynamic type of `E` as follows:
  - If the type of `E` is a reference type, `D` is the run-time type of the instance reference by `E`.
  - If the type of `E` is a nullable type, `D` is the underlying type of that nullable type.

- If the type of `E` is a non-nullable value type, `D` is the type of `E`.

- The result of the operation depends on `D` and `T` as follows:
  - If `T` is a reference type, the result is true if `D` and `T` are the same type, if `D` is a reference type and an implicit reference conversion from `D` to `T` exists, or if `D` is a value type and a boxing conversion from `D` to `T` exists.
  - If `T` is a nullable type, the result is true if `D` is the underlying type of `T`.
  - If `T` is a non-nullable value type, the result is true if `D` and `T` are the same type.
  - Otherwise, the result is false.

Note that user defined conversions, are not considered by the `is` operator.

The as operator

The `as` operator is used to explicitly convert a value to a given reference type or nullable type. Unlike a cast expression (Cast expressions), the `as` operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is `null`.

In an operation of the form `E as T`, `E` must be an expression and `T` must be a reference type, a type parameter known to be a reference type, or a nullable type. Furthermore, at least one of the following must be true, or otherwise a compile-time error occurs:

- An identity (Identity conversion), implicit nullable (Implicit nullable conversions), implicit reference (Implicit reference conversions), boxing (Boxing conversions), explicit nullable (Explicit nullable conversions), explicit reference (Explicit reference conversions), or unboxing (Unboxing conversions) conversion exists from `E` to `T`.
- The type of `E` or `T` is an open type.
- `E` is the `null` literal.

If the compile-time type of `E` is not `dynamic`, the operation `E as T` produces the same result as

```
E is T ? (T)(E) : (T)null
```

except that `E` is only evaluated once. The compiler can be expected to optimize `E as T` to perform at most one dynamic type check as opposed to the two dynamic type checks implied by the expansion above.

If the compile-time type of `E` is `dynamic`, unlike the cast operator the `as` operator is not dynamically bound (Dynamic binding). Therefore the expansion in this case is:

```
E is T ? (T)(object)(E) : (T)null
```

Note that some conversions, such as user defined conversions, are not possible with the `as` operator and should instead be performed using cast expressions.

In the example

```
class X
{

    public string F(object o) {
        return o as string;        // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;             // Ok, T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;             // Error, U is unconstrained
    }
}
```

the type parameter `T` of `G` is known to be a reference type, because it has the class constraint. The type parameter `U` of `H` is not however; hence the use of the `as` operator in `H` is disallowed.

Logical operators

The `&`, `^`, and `|` operators are called the logical operators.

```
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
```

If an operand of a logical operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x op y`, where `op` is one of the logical operators, overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

Integer logical operators

The predefined integer logical operators are:

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

The `&` operator computes the bitwise logical `AND` of the two operands, the `|` operator computes the bitwise logical `OR` of the two operands, and the `^` operator computes the bitwise logical exclusive `OR` of the two operands. No overflows are possible from these operations.

Enumeration logical operators

Every enumeration type `E` implicitly provides the following predefined logical operators:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and `op` is one of the logical operators, is exactly the same as evaluating `(E)((U)x op (U)y)`. In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

Boolean logical operators

The predefined boolean logical operators are:

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

The result of `x & y` is `true` if both `x` and `y` are `true`. Otherwise, the result is `false`.

The result of `x | y` is `true` if either `x` or `y` is `true`. Otherwise, the result is `false`.

The result of `x ^ y` is `true` if `x` is `true` and `y` is `false`, or `x` is `false` and `y` is `true`. Otherwise, the result is `false`. When the operands are of type `bool`, the `^` operator computes the same result as the `!=` operator.

Nullable boolean logical operators

The nullable boolean type `bool?` can represent three values, `true`, `false`, and `null`, and is conceptually similar to the three-valued type used for boolean expressions in SQL. To ensure that the results produced by the `&` and `|` operators for `bool?` operands are consistent with SQL's three-valued logic, the following predefined operators are provided:

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

The following table lists the results produced by these operators for all combinations of the values `true`, `false`,

and `null`.

`x`	`y`	`x & y`	`x | y`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`true`	`null`	`null`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`
`false`	`null`	`false`	`null`
`null`	`true`	`null`	`true`
`null`	`false`	`false`	`null`
`null`	`null`	`null`	`null`

Conditional logical operators

The `&&` and `||` operators are called the conditional logical operators. They are also called the "short-circuiting" logical operators.

```
conditional_and_expression
    : inclusive_or_expression
    | conditional_and_expression '&&' inclusive_or_expression
    ;

conditional_or_expression
    : conditional_and_expression
    | conditional_or_expression '||' conditional_and_expression
    ;
```

The `&&` and `||` operators are conditional versions of the `&` and `|` operators:

- The operation `x && y` corresponds to the operation `x & y`, except that `y` is evaluated only if `x` is not `false`.
- The operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is not `true`.

If an operand of a conditional logical operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

An operation of the form `x && y` or `x || y` is processed by applying overload resolution (Binary operator overload resolution) as if the operation was written `x & y` or `x | y`. Then,

- If overload resolution fails to find a single best operator, or if overload resolution selects one of the predefined integer logical operators, a binding-time error occurs.
- Otherwise, if the selected operator is one of the predefined boolean logical operators (Boolean logical operators) or nullable boolean logical operators (Nullable boolean logical operators), the operation is

processed as described in Boolean conditional logical operators.

- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in User-defined conditional logical operators.

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 User-defined conditional logical operators.

Boolean conditional logical operators

When the operands of `&&` or `||` are of type `bool`, or when the operands are of types that do not define an applicable `operator &` or `operator |`, but do define implicit conversions to `bool`, the operation is processed as follows:

- The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.
- The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation.

User-defined conditional logical operators

When the operands of `&&` or `||` are of types that declare an applicable user-defined `operator &` or `operator |`, both of the following must be true, where `T` is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator must be `T`. In other words, the operator must compute the logical `AND` or the logical `OR` of two operands of type `T`, and must return a result of type `T`.
- `T` must contain declarations of `operator true` and `operator false`.

A binding-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||` operation is evaluated by combining the user-defined `operator true` or `operator false` with the selected user-defined operator:

- The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an invocation of the `operator false` declared in `T`, and `T.&(x, y)` is an invocation of the selected `operator &`. In other words, `x` is first evaluated and `operator false` is invoked on the result to determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator &` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an invocation of the `operator true` declared in `T`, and `T.|(x,y)` is an invocation of the selected `operator|`. In other words, `x` is first evaluated and `operator true` is invoked on the result to determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator |` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.

In either of these operations, the expression given by `x` is only evaluated once, and the expression given by `y` is either not evaluated or evaluated exactly once.

For an example of a type that implements `operator true` and `operator false`, see Database boolean type.

The null coalescing operator

The `??` operator is called the null coalescing operator.

```
null_coalescing_expression
    : conditional_or_expression
    | conditional_or_expression '??' null_coalescing_expression
    ;
```

A null coalescing expression of the form `a ?? b` requires `a` to be of a nullable type or reference type. If `a` is non-null, the result of `a ?? b` is `a`; otherwise, the result is `b`. The operation evaluates `b` only if `a` is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ?? b ?? c` is evaluated as `a ?? (b ?? c)`. In general terms, an expression of the form `E1 ?? E2 ?? ... ?? En` returns the first of the operands that is non-null, or null if all operands are null.

The type of the expression `a ?? b` depends on which implicit conversions are available on the operands. In order of preference, the type of `a ?? b` is `A0`, `A`, or `B`, where `A` is the type of `a` (provided that `a` has a type), `B` is the type of `b` (provided that `b` has a type), and `A0` is the underlying type of `A` if `A` is a nullable type, or `A` otherwise. Specifically, `a ?? b` is processed as follows:

- If `A` exists and is not a nullable type or a reference type, a compile-time error occurs.
- If `b` is a dynamic expression, the result type is `dynamic`. At run-time, `a` is first evaluated. If `a` is not null, `a` is converted to dynamic, and this becomes the result. Otherwise, `b` is evaluated, and this becomes the result.
- Otherwise, if `A` exists and is a nullable type and an implicit conversion exists from `b` to `A0`, the result type is `A0`. At run-time, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type `A0`, and this becomes the result. Otherwise, `b` is evaluated and converted to type `A0`, and this becomes the result.
- Otherwise, if `A` exists and an implicit conversion exists from `b` to `A`, the result type is `A`. At run-time, `a` is first evaluated. If `a` is not null, `a` becomes the result. Otherwise, `b` is evaluated and converted to type `A`, and this becomes the result.
- Otherwise, if `b` has a type `B` and an implicit conversion exists from `a` to `B`, the result type is `B`. At run-time, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type `A0` (if `A` exists and is nullable) and converted to type `B`, and this becomes the result. Otherwise, `b` is evaluated and becomes the result.
- Otherwise, `a` and `b` are incompatible, and a compile-time error occurs.

Conditional operator

The `?:` operator is called the conditional operator. It is at times also called the ternary operator.

```
conditional_expression
    : null_coalescing_expression
    | null_coalescing_expression '?' expression ':' expression
    ;
```

A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the operation. A conditional expression never evaluates both `x` and `y`.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

The first operand of the `?:` operator must be an expression that can be implicitly converted to `bool`, or an expression of a type that implements `operator true`. If neither of these requirements is satisfied, a compile-time error occurs.

The second and third operands, `x` and `y`, of the `?:` operator control the type of the conditional expression.

- If `x` has type `X` and `y` has type `Y` then
  - If an implicit conversion (Implicit conversions) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of the conditional expression.
  - If an implicit conversion (Implicit conversions) exists from `Y` to `X`, but not from `X` to `Y`, then `X` is the type of the conditional expression.
  - Otherwise, no expression type can be determined, and a compile-time error occurs.
- If only one of `x` and `y` has a type, and both `x` and `y`, of are implicitly convertible to that type, then that is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

The run-time processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- First, `b` is evaluated, and the `bool` value of `b` is determined:
  - If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is performed to produce a `bool` value.
  - Otherwise, the `operator true` defined by the type of `b` is invoked to produce a `bool` value.
- If the `bool` value produced by the step above is `true`, then `x` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

Anonymous function expressions

An ***anonymous function*** is an expression that represents an "in-line" method definition. An anonymous function does not have a value or type in and of itself, but is convertible to a compatible delegate or expression tree type. The evaluation of an anonymous function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method which the anonymous function defines. If it is an expression tree type, the conversion evaluates to an expression tree which represents the structure of the method as an object structure.

For historical reasons there are two syntactic flavors of anonymous functions, namely *lambda_expression*s and *anonymous_method_expression*s. For almost all purposes, *lambda_expression*s are more concise and expressive than *anonymous_method_expression*s, which remain in the language for backwards compatibility.

```
lambda_expression
    : anonymous_function_signature '=>' anonymous_function_body
    ;

anonymous_method_expression
    : 'delegate' explicit_anonymous_function_signature? block
    ;

anonymous_function_signature
    : explicit_anonymous_function_signature
    | implicit_anonymous_function_signature
    ;

explicit_anonymous_function_signature
    : '(' explicit_anonymous_function_parameter_list? ')'
    ;

explicit_anonymous_function_parameter_list
    : explicit_anonymous_function_parameter (',' explicit_anonymous_function_parameter)*
    ;

explicit_anonymous_function_parameter
    : anonymous_function_parameter_modifier? type identifier
    ;

anonymous_function_parameter_modifier
    : 'ref'
    | 'out'
    ;

implicit_anonymous_function_signature
    : '(' implicit_anonymous_function_parameter_list? ')'
    | implicit_anonymous_function_parameter
    ;

implicit_anonymous_function_parameter_list
    : implicit_anonymous_function_parameter (',' implicit_anonymous_function_parameter)*
    ;

implicit_anonymous_function_parameter
    : identifier
    ;

anonymous_function_body
    : expression
    | block
    ;
```

The `=>` operator has the same precedence as assignment (`=`) and is right-associative.

An anonymous function with the `async` modifier is an async function and follows the rules described in Iterators.

The parameters of an anonymous function in the form of a *lambda_expression* can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types (Anonymous function conversions).

In an anonymous function with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

```
( param ) => expr
```

can be abbreviated to

```
param => expr
```

The parameter list of an anonymous function in the form of an *anonymous_method_expression* is optional. If given, the parameters must be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing `out` parameters.

A *block* body of an anonymous function is reachable (End points and reachability) unless the anonymous function occurs inside an unreachable statement.

Some examples of anonymous functions follow below:

```
x => x + 1                           // Implicitly typed, expression body
x => { return x + 1; }               // Implicitly typed, statement body
(int x) => x + 1                     // Explicitly typed, expression body
(int x) => { return x + 1; }         // Explicitly typed, statement body
(x, y) => x * y                      // Multiple parameters
() => Console.WriteLine()            // No parameters
async (t1,t2) => await t1 + await t2 // Async
delegate (int x) { return x + 1; }   // Anonymous method expression
delegate { return 1 + 1; }           // Parameter list omitted
```

The behavior of *lambda_expression*s and *anonymous_method_expression*s is the same except for the following points:

- *anonymous_method_expression*s permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of value parameters.
- *lambda_expression*s permit parameter types to be omitted and inferred whereas *anonymous_method_expression*s require parameter types to be explicitly stated.
- The body of a *lambda_expression* can be an expression or a statement block whereas the body of an *anonymous_method_expression* must be a statement block.
- Only *lambda_expression*s have conversions to compatible expression tree types (Expression tree types).

Anonymous function signatures

The optional *anonymous_function_signature* of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the *anonymous_function_body*. (Scopes) Together with the parameter list (if given) the anonymous-method-body constitutes a declaration space (Declarations). It is thus a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant or parameter whose scope includes the *anonymous_method_expression* or *lambda_expression*.

If an anonymous function has an *explicit_anonymous_function_signature*, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order. In contrast to method group conversions (Method group conversions), contra-variance of anonymous function parameter types is not supported. If an anonymous function does not have an *anonymous_function_signature*, then the set of compatible delegate types and expression tree types is restricted to those that have no `out` parameters.

Note that an *anonymous_function_signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous_function_signature* may be compatible with a delegate type whose parameter list contains a parameter array.

Note also that conversion to an expression tree type, even if compatible, may still fail at compile-time (Expression tree types).

Anonymous function bodies

The body (*expression* or *block*) of an anonymous function is subject to the following rules:

- If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature it can be converted to a delegate type or expression type having parameters (Anonymous function conversions), but the parameters cannot be accessed in the body.
- Except for `ref` or `out` parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a `ref` or `out` parameter.
- When the type of `this` is a struct type, it is a compile-time error for the body to access `this`. This is true whether the access is explicit (as in `this.x`) or implicit (as in `x` where `x` is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
- The body has access to the outer variables (Outer variables) of the anonymous function. Access of an outer variable will reference the instance of the variable that is active at the time the *lambda_expression* or *anonymous_method_expression* is evaluated (Evaluation of anonymous function expressions).
- It is a compile-time error for the body to contain a `goto` statement, `break` statement, or `continue` statement whose target is outside the body or within the body of a contained anonymous function.
- A `return` statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member. An expression specified in a `return` statement must be implicitly convertible to the return type of the delegate type or expression tree type to which the nearest enclosing *lambda_expression* or *anonymous_method_expression* is converted (Anonymous function conversions).

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the *lambda_expression* or *anonymous_method_expression*. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements must be of a form reserved for compiler use.

Overload resolution and anonymous functions

Anonymous functions in an argument list participate in type inference and overload resolution. Please refer to Type inference and Overload resolution for the exact rules.

The following example illustrates the effect of anonymous functions on overload resolution.

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

The `ItemList<T>` class has two `Sum` methods. Each takes a `selector` argument, which extracts the value to sum over from a list item. The extracted value can be either an `int` or a `double` and the resulting sum is likewise either an `int` or a `double`.

The `Sum` methods could for example be used to compute sums from a list of detail lines in an order.

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

In the first invocation of `orderDetails.Sum`, both `Sum` methods are applicable because the anonymous function `d => d. UnitCount` is compatible with both `Func<Detail,int>` and `Func<Detail,double>`. However, overload resolution picks the first `Sum` method because the conversion to `Func<Detail,int>` is better than the conversion to `Func<Detail,double>`.

In the second invocation of `orderDetails.Sum`, only the second `Sum` method is applicable because the anonymous function `d => d.UnitPrice * d.UnitCount` produces a value of type `double`. Thus, overload resolution picks the second `Sum` method for that invocation.

Anonymous functions and dynamic binding

An anonymous function cannot be a receiver, argument or operand of a dynamically bound operation.

Outer variables

Any local variable, value parameter, or parameter array whose scope includes the *lambda_expression* or *anonymous_method_expression* is called an ***outer variable*** of the anonymous function. In an instance function member of a class, the `this` value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

Captured outer variables

When an outer variable is referenced by an anonymous function, the outer variable is said to have been ***captured*** by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated (Local variables). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

In the example

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

the local variable `x` is captured by the anonymous function, and the lifetime of `x` is extended at least until the delegate returned from `F` becomes eligible for garbage collection (which doesn't happen until the very end of the program). Since each invocation of the anonymous function operates on the same instance of `x`, the output of the example is:

```
1
2
3
```

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable (Fixed and moveable variables), but is instead considered to be a moveable variable. Thus any `unsafe` code that takes the address of a captured outer variable must first use the `fixed` statement to fix the variable.

Note that unlike an uncaptured variable, a captured local variable can be simultaneously exposed to multiple threads of execution.

Instantiation of local variables

A local variable is considered to be ***instantiated*** when execution enters the scope of the variable. For example, when the following method is invoked, the local variable `x` is instantiated and initialized three times—once for each iteration of the loop.

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

However, moving the declaration of `x` outside the loop results in a single instantiation of `x`:

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

When not captured, there is no way to observe exactly how often a local variable is instantiated—because the lifetimes of the instantiations are disjoint, it is possible for each instantiation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

The example

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produces the output:

```
1
3
5
```

However, when the declaration of `x` is moved outside the loop:

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

the output is:

```
5
5
5
```

If a for-loop declares an iteration variable, that variable itself is considered to be declared outside of the loop. Thus, if the example is changed to capture the iteration variable itself:

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

only one instance of the iteration variable is captured, which produces the output:

```
3
3
3
```

It is possible for anonymous function delegates to share some captured variables yet have separate instances of others. For example, if `F` is changed to

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

the three delegates capture the same instance of `x` but separate instances of `y`, and the output is:

```
1 1
2 1
3 1
```

Separate anonymous functions can capture the same instance of an outer variable. In the example:

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

the two anonymous functions capture the same instance of the local variable `x`, and they can thus "communicate"

through that variable. The output of the example is:

```
5
10
```

Evaluation of anonymous function expressions

An anonymous function `F` must always be converted to a delegate type `D` or an expression tree type `E`, either directly or through the execution of a delegate creation expression `new D(F)`. This conversion determines the result of the anonymous function, as described in Anonymous function conversions.

Query expressions

Query expressions provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

```
query_expression
    : from_clause query_body
    ;

from_clause
    : 'from' type? identifier 'in' expression
    ;

query_body
    : query_body_clauses? select_or_group_clause query_continuation?
    ;

query_body_clauses
    : query_body_clause
    | query_body_clauses query_body_clause
    ;

query_body_clause
    : from_clause
    | let_clause
    | where_clause
    | join_clause
    | join_into_clause
    | orderby_clause
    ;

let_clause
    : 'let' identifier '=' expression
    ;

where_clause
    : 'where' boolean_expression
    ;

join_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression
    ;

join_into_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression 'into' identifier
    ;

orderby_clause
    : 'orderby' orderings
    ;

orderings
    : ordering (',' ordering)*
```

```
    ;

ordering
    : expression ordering_direction?
    ;

ordering_direction
    : 'ascending'
    | 'descending'
    ;

select_or_group_clause
    : select_clause
    | group_clause
    ;

select_clause
    : 'select' expression
    ;

group_clause
    : 'group' expression 'by' expression
    ;

query_continuation
    : 'into' identifier query_body
    ;
```

A query expression begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let`, `where`, `join` or `orderby` clauses. Each `from` clause is a generator introducing a ***range variable*** which ranges over the elements of a ***sequence***. Each `let` clause introduces a range variable representing a value computed by means of previous range variables. Each `where` clause is a filter that excludes items from the result. Each `join` clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each `orderby` clause reorders items according to specified criteria.The final `select` or `group` clause specifies the shape of the result in terms of the range variables. Finally, an `into` clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

Ambiguities in query expressions

Query expressions contain a number of "contextual keywords", i.e., identifiers that have special meaning in a given context. Specifically these are `from`, `where`, `join`, `on`, `equals`, `into`, `let`, `orderby`, `ascending`, `descending`, `select`, `group` and `by`. In order to avoid ambiguities in query expressions caused by mixed use of these identifiers as keywords or simple names, these identifiers are considered keywords when occurring anywhere within a query expression.

For this purpose, a query expression is any expression that starts with "`from identifier`" followed by any token except "`;`", "`=`" or "`,`".

In order to use these words as identifiers within a query expression, they can be prefixed with "`@`" (Identifiers).

Query expression translation

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the *query expression pattern* (The query expression pattern). Specifically, query expressions are translated into invocations of methods named `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast`.These methods are expected to have particular signatures and result types, as described in The query expression pattern. These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type

binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of application: each section assumes that the translations in the preceding sections have been performed exhaustively, and once exhausted, a section will not later be revisited in the processing of the same query expression.

Assignment to range variables is not allowed in query expressions. However a C# implementation is permitted to not always enforce this restriction, since this may sometimes not be possible with the syntactic translation scheme presented here.

Certain translations inject range variables with transparent identifiers denoted by `*`. The special properties of transparent identifiers are discussed further in Transparent identifiers.

Select and groupby clauses with continuations

A query expression with a continuation

```
from ... into x ...
```

is translated into

```
from x in ( from ... ) ...
```

The translations in the following sections assume that queries have no `into` continuations.

The example

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

is translated into

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

the final translation of which is

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

Explicit range variable types

A `from` clause that explicitly specifies a range variable type

```
from T x in e
```

is translated into

```
from x in ( e ) . Cast < T > ( )
```

A `join` clause that explicitly specifies a range variable type

```
join T x in e on k1 equals k2
```

is translated into

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

The translations in the following sections assume that queries have no explicit range variable types.

The example

```
from Customer c in customers
where c.City == "London"
select c
```

is translated into

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

the final translation of which is

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Explicit range variable types are useful for querying collections that implement the non-generic `IEnumerable` interface, but not the generic `IEnumerable<T>` interface. In the example above, this would be the case if `customers` were of type `ArrayList`.

Degenerate query expressions

A query expression of the form

```
from x in e select x
```

is translated into

```
( e ) . Select ( x => x )
```

The example

```
from c in customers
select c
```

is translated into

```
customers.Select(c => c)
```

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. It is important however to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling `Select` on the source. It is then up to the implementers of `Select` and other query operators to ensure that these methods never return the source object itself.

From, let, where, join and orderby clauses

A query expression with a second `from` clause followed by a `select` clause

```
from x1 in e1
from x2 in e2
select v
```

is translated into

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

A query expression with a second `from` clause followed by something other than a `select` clause:

```
from x1 in e1
from x2 in e2
...
```

is translated into

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
```

A query expression with a `let` clause

```
from x in e
let y = f
...
```

is translated into

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

A query expression with a `where` clause

```
from x in e
where f
...
```

is translated into

```
from x in ( e ) . Where ( x => f )
...
```

A query expression with a `join` clause without an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

is translated into

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

A query expression with a `join` clause without an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

is translated into

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

A query expression with a `join` clause with an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

is translated into

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

A query expression with a `join` clause with an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

is translated into

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
...
```

A query expression with an `orderby` clause

```
from x in e
orderby k1 , k2 , ..., kn
...
```

is translated into

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
... .
ThenBy ( x => kn )
...
```

If an ordering clause specifies a `descending` direction indicator, an invocation of `OrderByDescending` or `ThenByDescending` is produced instead.

The following translations assume that there are no `let`, `where`, `join` or `orderby` clauses, and no more than the one initial `from` clause in each query expression.

The example

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
customers.
SelectMany(c => c.Orders,
     (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

the final translation of which is

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

is translated into

```
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

the final translation of which is

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

is translated into

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

is translated into

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

the final translation of which is

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

where `x` and `y` are compiler generated identifiers that are otherwise invisible and inaccessible.

The example

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

has the final translation

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

Select clauses

A query expression of the form

```
from x in e select v
```

is translated into

```
( e ) . Select ( x => v )
```

except when v is the identifier x, the translation is simply

```
( e )
```

For example

```
from c in customers.Where(c => c.City == "London")
select c
```

is simply translated into

```
customers.Where(c => c.City == "London")
```

Groupby clauses

A query expression of the form

```
from x in e group v by k
```

is translated into

```
( e ) . GroupBy ( x => k , x => v )
```

except when v is the identifier x, the translation is

```
( e ) . GroupBy ( x => k )
```

The example

```
from c in customers
group c.Name by c.Country
```

is translated into

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

Transparent identifiers

Certain translations inject range variables with ***transparent identifiers*** denoted by `*`. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

- When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.
- When a member with a transparent identifier is in scope, the members of that member are in scope as well.
- When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.
- In the translation steps described above, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and show how transparent identifiers can be translated away.

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

which is further translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

which, when transparent identifiers are erased, is equivalent to

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

is translated into

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

which is further reduced to

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

the final translation of which is

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
    (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
    (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

where `x`, `y`, and `z` are compiler generated identifiers that are otherwise invisible and inaccessible.

The query expression pattern

The ***Query expression pattern*** establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the methods

of the pattern can be implemented as instance methods or as extension methods because the two have the same invocation syntax, and the methods can request delegates or expression trees because anonymous functions are convertible to both.

The recommended shape of a generic type `C<T>` that supports the query expression pattern is shown below. A generic type is used in order to illustrate the proper relationships between parameter and result types, but it is possible to implement the pattern for non-generic types as well.

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

The methods above use the generic delegate types `Func<T1,R>` and `Func<T1,T2,R>`, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between `C<T>` and `O<T>` which ensures that the `ThenBy` and `ThenByDescending` methods are available only on the result of an `OrderBy` or `OrderByDescending`. Also notice the recommended shape of the result of `GroupBy` -- a sequence of sequences, where each inner sequence has an additional `Key` property.

The `System.Linq` namespace provides an implementation of the query operator pattern for any type that

implements the `System.Collections.Generic.IEnumerable<T>` interface.

Assignment operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

```
assignment
    : unary_expression assignment_operator expression
    ;

assignment_operator
    : '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | right_shift_assignment
    ;
```

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The `=` operator is called the ***simple assignment operator***. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in Field-like events). The simple assignment operator is described in Simple assignment.

The assignment operators other than the `=` operator are called the ***compound assignment operators***. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in Compound assignment.

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 Event assignment.

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a = b = c` is evaluated as `a = (b = c)`.

Simple assignment

The `=` operator is called the simple assignment operator.

If the left operand of a simple assignment is of the form `E.P` or `E[Ei]` where `E` has the compile-time type `dynamic`, then the assignment is dynamically bound (Dynamic binding). In this case the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at run-time based on the run-time type of `E`.

In a simple assignment, the right operand must be an expression that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a `set` accessor. If this is not the case, a binding-time error occurs.

The run-time processing of a simple assignment of the form `x = y` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (Implicit conversions).
  - 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 (Implicit reference conversions) 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 (Implicit conversions).
  - The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

The array co-variance rules (Array covariance) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, assignment to an array element of a *reference_type* requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;               // Ok
oa[1] = "Hello";            // Ok
oa[2] = new ArrayList();    // ArrayTypeMismatchException
```

the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance of `ArrayList` cannot be stored in an element of a `string[]`.

When a property or indexer declared in a *struct_type* is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a binding-time error occurs. Because of Member access, the same rule also applies to fields.

Given the declarations:

```
struct Point
{
    int x, y;

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

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

struct Rectangle
{
    Point a, b;

    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }

    public Point A {
        get { return a; }
        set { a = value; }
    }

    public Point B {
        get { return b; }
        set { b = value; }
    }
}
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid, since `r.A` and `r.B` are not variables.

Compound assignment

If the left operand of a compound assignment is of the form `E.P` or `E[Ei]` where `E` has the compile-time type

`dynamic`, then the assignment is dynamically bound (Dynamic binding). In this case the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at run-time based on the run-time type of `E`.

An operation of the form `x op= y` is processed by applying binary operator overload resolution (Binary operator overload resolution) as if the operation was written `x op y`. Then,

- If the return type of the selected operator is implicitly convertible to the type of `x`, the operation is evaluated as `x = x op y`, except that `x` is evaluated only once.
- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is explicitly convertible to the type of `x`, and if `y` is implicitly convertible to the type of `x` or the operator is a shift operator, then the operation is evaluated as `x = (T)(x op y)`, where `T` is the type of `x`, except that `x` is evaluated only once.
- Otherwise, the compound assignment is invalid, and a binding-time error occurs.

The term "evaluated only once" means that in the evaluation of `x op y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. For example, in the assignment `A()[B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are methods returning `int`, the methods are invoked only once, in the order `A`, `B`, `C`.

When the left operand of a compound assignment is a property access or indexer access, the property or indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a binding-time error occurs.

The second rule above permits `x op= y` to be evaluated as `x = (T)(x op y)` in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the predefined operators produce a result of type `int`, as described in Binary numeric promotions. Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that `x op= y` is permitted if both of `x op y` and `x = y` are permitted. In the example

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;             // Ok
b += 1000;          // Error, b = 1000 not permitted
b += i;             // Error, b = i not permitted
b += (byte)i;       // Ok

ch += 1;            // Error, ch = 1 not permitted
ch += (char)1;      // Ok
```

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

This also means that compound assignment operations support lifted operations. In the example

```
int? i = 0;
i += 1;             // Ok
```

the lifted operator `+(int?,int?)` is used.

Event assignment

If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as follows:

- The instance expression, if any, of the event access is evaluated.
- The right operand of the `+=` or `-=` operator is evaluated, and, if required, converted to the type of the left operand through an implicit conversion (Implicit conversions).
- 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* (Expression statements).

Expression

An *expression* is either a *non_assignment_expression* or an *assignment*.

```
expression
    : non_assignment_expression
    | assignment
    ;

non_assignment_expression
    : conditional_expression
    | lambda_expression
    | query_expression
    ;
```

Constant expressions

A *constant_expression* is an expression that can be fully evaluated at compile-time.

```
constant_expression
    : expression
    ;
```

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `object`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables
- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions
- Default value expressions
- Nameof expressions
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

```
class C {
    const object i = 5;         // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of i is an error because a boxing conversion is required. The initialization of str is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (Constant expressions).

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 (Constants).
- Enumeration member declarations (Enum members).
- Default arguments of formal parameter lists (Method parameters)
- `case` labels of a `switch` statement (The switch statement).
- `goto case` statements (The goto statement).
- Dimension lengths in an array creation expression (Array creation expressions) that includes an initializer.
- Attributes (Attributes).

An implicit constant expression conversion (Implicit constant expression conversions) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

Boolean expressions

A *boolean_expression* is an expression that yields a result of type `bool`; either directly or through application of `operator true` in certain contexts as specified in the following.

```
boolean_expression
    : expression
    ;
```

The controlling conditional expression of an *if_statement* (The if statement), *while_statement* (The while

statement), *do_statement* (The do statement), or *for_statement* (The for statement) is a *boolean_expression*. The controlling conditional expression of the `?:` operator (Conditional operator) follows the same rules as a *boolean_expression*, but for reasons of operator precedence is classified as a *conditional_or_expression*.

A *boolean_expression* `E` is required to be able to produce a value of type `bool`, as follows:

- If `E` is implicitly convertible to `bool` then at runtime that implicit conversion is applied.
- Otherwise, unary operator overload resolution (Unary operator overload resolution) is used to find a unique best implementation of operator `true` on `E`, and that implementation is applied at runtime.
- If no such operator is found, a binding-time error occurs.

The `DBBool` struct type in Database boolean type provides an example of a type that implements `operator true` and `operator false`.

Statements

1/13/2018 • 54 minutes to read • Edit Online

C# provides a variety of statements. Most of these statements will be familiar to developers who have programmed in C and C++.

```
statement
    : labeled_statement
    | declaration_statement
    | embedded_statement
    ;

embedded_statement
    : block
    | empty_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | try_statement
    | checked_statement
    | unchecked_statement
    | lock_statement
    | using_statement
    | yield_statement
    | embedded_statement_unsafe
    ;
```

The *embedded_statement* nonterminal is used for statements that appear within other statements. The use of *embedded_statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. The example

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

results in a compile-time error because an `if` statement requires an *embedded_statement* rather than a *statement* for its if branch. If this code were permitted, then the variable `i` would be declared, but it could never be used. Note, however, that by placing `i`'s declaration in a block, the example is valid.

End points and reachability

Every statement has an ***end point***. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be ***reachable***. Conversely, if there is no possibility that a statement will be executed, the statement is said to be ***unreachable***.

In the example

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
    Label:
    Console.WriteLine("reachable");
}
```

the second invocation of `Console.WriteLine` is unreachable because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (Constant expressions) that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

the boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. As the constant expression is evaluated at compile-time, producing the value `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a local variable

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be executed.

The *block* of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

In the example

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

the reachability of the second `Console.WriteLine` is determined as follows:

- The first `Console.WriteLine` expression statement is reachable because the block of the `F` method is reachable.
- The end point of the first `Console.WriteLine` expression statement is reachable because that statement is reachable.
- The `if` statement is reachable because the end point of the first `Console.WriteLine` expression statement is reachable.

- The second `Console.WriteLine` expression statement is reachable because the boolean expression of the `if` statement does not have the constant value `false`.

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the `switch` statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it typically is an indication that a `return` statement is missing.

Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

```
block
    : '{' statement_list? '}'
    ;
```

A *block* consists of an optional *statement_list* (Statement lists), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements (Declaration statements). The scope of a local variable or constant declared in a block is the block.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A *block* that contains one or more `yield` statements (The yield statement) is called an iterator block. Iterator blocks are used to implement function members as iterators (Iterators). Some additional restrictions apply to iterator blocks:

- It is a compile-time error for a `return` statement to appear in an iterator block (but `yield return` statements are permitted).
- It is a compile-time error for an iterator block to contain an unsafe context (Unsafe contexts). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

Statement lists

A ***statement list*** consists of one or more statements written in sequence. Statement lists occur in *block*s (Blocks) and in *switch_block*s (The switch statement).

```
statement_list
    : statement+
    ;
```

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable `goto` statement.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

The empty statement

An *empty_statement* does nothing.

```
empty_statement
    : ';'
    ;
```

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

An empty statement can be used when writing a `while` statement with a null body:

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

Also, an empty statement can be used to declare a label just before the closing "`}`" of a block:

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

Labeled statements

A *labeled_statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

```
labeled_statement
    : identifier ':' statement
    ;
```

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from `goto` statements (The goto statement) within the scope of the label. This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

```
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

is valid and uses the name `x` as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable `goto` statement. (Exception: If a `goto` statement is inside a `try` that includes a `finally` block, and the labeled statement is outside the `try`, and the end point of the `finally` block is unreachable, then the labeled statement is not reachable from that `goto` statement.)

Declaration statements

A *declaration_statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

```
declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    ;
```

Local variable declarations

A *local_variable_declaration* declares one or more local variables.

```
local_variable_declaration
    : local_variable_type local_variable_declarators
    ;

local_variable_type
    : type
    | 'var'
    ;

local_variable_declarators
    : local_variable_declarator
    | local_variable_declarators ',' local_variable_declarator
    ;

local_variable_declarator
    : identifier
    | identifier '=' local_variable_initializer
    ;

local_variable_initializer
    : expression
    | array_initializer
    | local_variable_initializer_unsafe
    ;
```

The *local_variable_type* of a *local_variable_declaration* either directly specifies the type of the variables introduced by the declaration, or indicates with the identifier `var` that the type should be inferred based on an initializer. The type is followed by a list of *local_variable_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.

In the context of a local variable declaration, the identifier var acts as a contextual keyword (Keywords).When the *local_variable_type* is specified as `var` and no type named `var` is in scope, the declaration is an ***implicitly typed local variable declaration***, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

- The *local_variable_declaration* cannot include multiple *local_variable_declarator*s.
- The *local_variable_declarator* must include a *local_variable_initializer*.
- The *local_variable_initializer* must be an *expression*.
- The initializer *expression* must have a compile-time type.
- The initializer *expression* cannot refer to the declared variable itself

The following are examples of incorrect implicitly typed local variable declarations:

```
var x;               // Error, no initializer to infer type from
var y = {1, 2, 3};   // Error, array initializer not permitted
var z = null;        // Error, null does not have a type
var u = x => x + 1;  // Error, anonymous functions do not have a type
var v = v++;         // Error, initializer cannot refer to variable itself
```

The value of a local variable is obtained in an expression using a *simple_name* (Simple names), and the value of a local variable is modified using an *assignment* (Assignment operators). A local variable must be definitely assigned (Definite assignment) at each location where its value is obtained.

The scope of a local variable declared in a *local_variable_declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local_variable_declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
void F() {
    int x = 1, y, z = x * 2;
}
```

corresponds exactly to

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

In an implicitly typed local variable declaration, the type of the local variable being declared is taken to be the same as the type of the expression used to initialize the variable. For example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Local constant declarations

A *local_constant_declaration* declares one or more local constants.

```
local_constant_declaration
    : 'const' type constant_declarators
    ;

constant_declarators
    : constant_declarator (',' constant_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* (Constant expressions) 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 (Constants).

The value of a local constant is obtained in an expression using a *simple_name* (Simple names).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its *constant_declarator*. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

Expression statements

An *expression_statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

```
expression_statement
    : statement_expression ';'
    ;

statement_expression
    : invocation_expression
    | null_conditional_invocation_expression
    | object_creation_expression
    | assignment
    | post_increment_expression
    | post_decrement_expression
    | pre_increment_expression
    | pre_decrement_expression
    | await_expression
    ;
```

Not all expressions are permitted as statements. In particular, expressions such as `x + y` and `x == 1` that merely compute a value (which will be discarded), are not permitted as statements.

Execution of an *expression_statement* evaluates the contained expression and then transfers control to the end point of the *expression_statement*. The end point of an *expression_statement* is reachable if that *expression_statement* is reachable.

Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

```
selection_statement
    : if_statement
    | switch_statement
    ;
```

The if statement

The `if` statement selects a statement for execution based on the value of a boolean expression.

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Thus, an `if` statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

An `if` statement is executed as follows:

- The *boolean_expression* (Boolean expressions) is evaluated.
- If the boolean expression yields `true`, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is not present, control is transferred to the end point of the `if` statement.

The first embedded statement of an `if` statement is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `false`.

The second embedded statement of an `if` statement, if present, is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The end point of an `if` statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an `if` statement with no `else` part is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The switch statement

The switch statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

```
switch_statement
    : 'switch' '(' expression ')' switch_block
    ;

switch_block
    : '{' switch_section* '}'
    ;

switch_section
    : switch_label+ statement_list
    ;

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_label*s followed by a *statement_list* (Statement lists).

The ***governing type*** of a `switch` statement is established by the switch expression.

- If the type of the switch expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string`, or an *enum_type*, or if it is the nullable type corresponding to one of these types, then that is the governing type of the `switch` statement.
- Otherwise, exactly one user-defined implicit conversion (User-defined conversions) must exist from the type of the switch expression to one of the following possible governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or, a nullable type corresponding to one of those types.
- Otherwise, if no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each `case` label must denote a value that is implicitly convertible (Implicit conversions) to the governing type of the `switch` statement. A compile-time error occurs if two or more `case` labels in the same `switch` statement specify the same constant value.

There can be at most one `default` label in a switch statement.

A `switch` statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a `case` label in the same `switch` statement is equal to the value of the switch expression, control is transferred to the statement list following the matched `case` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if a `default` label is present, control is transferred to the statement list following the `default` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if no `default` label is present, control is transferred to the end point of the `switch` statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. The example

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to "fall through" to the next switch section, and the example

```
switch (i) {
case 0:
    CaseZero();
case 1:
    CaseZeroOrOne();
default:
    CaseAny();
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit `goto case` or `goto default` statement must be used:

```
switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}
```

Multiple labels are permitted in a *switch_section*. The example

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}
```

is valid. The example does not violate the "no fall through" rule because the labels `case 2:` and `default:` are part of the same *switch_section*.

The "no fall through" rule prevents a common class of bugs that occur in C and C++ when `break` statements are accidentally omitted. In addition, because of this rule, the switch sections of a `switch` statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the `switch` statement above can be reversed without affecting the behavior of the statement:

```
switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}
```

The statement list of a switch section typically ends in a `break`, `goto case`, or `goto default` statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a `while` statement controlled by the boolean expression `true` is known to never reach its end point. Likewise, a `throw` or `return` statement always transfers control elsewhere and never reaches its end point. Thus, the following example is valid:

```
switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```

The governing type of a `switch` statement may be the type `string`. For example:

```
void DoCommand(string command) {
    switch (command.ToLower()) {
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidCommand(command);
        break;
    }
}
```

Like the string equality operators (String equality operators), the `switch` statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a `case` label constant.

When the governing type of a `switch` statement is `string`, the value `null` is permitted as a case label constant.

The *statement_lists* of a *switch_block* may contain declaration statements (Declaration statements). The scope of a local variable or constant declared in a switch block is the switch block.

The statement list of a given switch section is reachable if the `switch` statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a `case` label in the switch section.
- The switch expression is a constant value that doesn't match any `case` label, and the switch section contains the `default` label.
- A switch label of the switch section is referenced by a reachable `goto case` or `goto default` statement.

The end point of a `switch` statement is reachable if at least one of the following is true:

- The `switch` statement contains a reachable `break` statement that exits the `switch` statement.
- The `switch` statement is reachable, the switch expression is a non-constant value, and no `default` label is present.
- The `switch` statement is reachable, the switch expression is a constant value that doesn't match any `case` label, and no `default` label is present.

Iteration statements

Iteration statements repeatedly execute an embedded statement.

```
iteration_statement
    : while_statement
    | do_statement
    | for_statement
    | foreach_statement
    ;
```

The while statement

The `while` statement conditionally executes an embedded statement zero or more times.

```
while_statement
    : 'while' '(' boolean_expression ')' embedded_statement
    ;
```

A `while` statement is executed as follows:

- The *boolean_expression* (Boolean expressions) 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 (The break statement) 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 (The continue statement) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the `while` statement).

The embedded statement of a `while` statement is reachable if the `while` statement is reachable and the boolean expression does not have the constant value `false`.

The end point of a `while` statement is reachable if at least one of the following is true:

- The `while` statement contains a reachable `break` statement that exits the `while` statement.
- The `while` statement is reachable and the boolean expression does not have the constant value `true`.

The do statement

The `do` statement conditionally executes an embedded statement one or more times.

```
do_statement
    : 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
    ;
```

A `do` statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the *boolean_expression* (Boolean expressions) 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 (The break statement) 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 (The continue statement) may be used to transfer control to the end point of the embedded statement.

The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

The end point of a `do` statement is reachable if at least one of the following is true:

- The `do` statement contains a reachable `break` statement that exits the `do` statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value `true`.

The for statement

The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

```
for_statement
    : 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')' 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)*
    ;
```

The *for_initializer*, if present, consists of either a *local_variable_declaration* (Local variable declarations) or a list of *statement_expressions* (Expression statements) 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* (Boolean expressions).

The *for_iterator*, if present, consists of a list of *statement_expressions* (Expression statements) 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 (The break statement) 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 (The continue statement) may be used to transfer control to the end point of the embedded

statement (thus executing the *for_iterator* and performing another iteration of the `for` statement, starting with the *for_condition*).

The embedded statement of a `for` statement is reachable if one of the following is true:

- The `for` statement is reachable and no *for_condition* is present.
- The `for` statement is reachable and a *for_condition* is present and does not have the constant value `false`.

The end point of a `for` statement is reachable if at least one of the following is true:

- The `for` statement contains a reachable `break` statement that exits the `for` statement.
- The `for` statement is reachable and a *for_condition* is present and does not have the constant value `true`.

The foreach statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

```
foreach_statement
    : 'foreach' '(' local_variable_type identifier 'in' expression ')' embedded_statement
    ;
```

The *type* and *identifier* of a `foreach` statement declare the ***iteration variable*** of the statement. If the `var` identifier is given as the *local_variable_type*, and no type named `var` is in scope, the iteration variable is said to be an ***implicitly typed iteration variable***, and its type is taken to be the element type of the `foreach` statement, as specified below. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (via assignment or the `++` and `--` operators) or pass the iteration variable as a `ref` or `out` parameter.

In the following, for brevity, `IEnumerable`, `IEnumerator`, `IEnumerable<T>` and `IEnumerator<T>` refer to the corresponding types in the namespaces `System.Collections` and `System.Collections.Generic`.

The compile-time processing of a foreach statement first determines the ***collection type***, ***enumerator type*** and ***element type*** of the expression. This determination proceeds as follows:

- If the type `X` of *expression* is an array type then there is an implicit reference conversion from `X` to the `IEnumerable` interface (since `System.Array` implements this interface). The ***collection type*** is the `IEnumerable` interface, the ***enumerator type*** is the `IEnumerator` interface and the ***element type*** is the element type of the array type `X`.
- If the type `X` of *expression* is `dynamic` then there is an implicit conversion from *expression* to the `IEnumerable` interface (Implicit dynamic conversions). The ***collection type*** is the `IEnumerable` interface and the ***enumerator type*** is the `IEnumerator` interface. If the `var` identifier is given as the *local_variable_type* then the ***element type*** is `dynamic`, otherwise it is `object`.
- Otherwise, determine whether the type `X` has an appropriate `GetEnumerator` method:
  - Perform member lookup on the type `X` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
  - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described below. It is recommended that a warning be issued if overload resolution produces anything except an

unambiguous public instance method or no applicable methods.
    - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
    - Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
    - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, check for an enumerable interface:
    - If among all the types `Ti` for which there is an implicit conversion from `X` to `IEnumerable<Ti>`, there is a unique type `T` such that `T` is not `dynamic` and for all the other `Ti` there is an implicit conversion from `IEnumerable<T>` to `IEnumerable<Ti>`, then the ***collection type*** is the interface `IEnumerable<T>`, the ***enumerator type*** is the interface `IEnumerator<T>`, and the ***element type*** is `T`.
    - Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.
    - Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the ***collection type*** is this interface, the ***enumerator type*** is the interface `System.Collections.IEnumerator`, and the ***element type*** is `object`.
    - Otherwise, an error is produced and no further steps are taken.

The above steps, if successful, unambiguously produce a collection type `C`, enumerator type `E` and element type `T`. A foreach statement of the form

```
foreach (V v in x) embedded_statement
```

is then expanded to:

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded_statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

The variable `e` is not visible to or accessible to the expression `x` or the embedded statement or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (Explicit conversions) from `T` (the element type) to `V` (the *local_variable_type* in the foreach statement), an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at run-time.

An implementation is permitted to implement a given foreach-statement differently, e.g. for performance reasons, as long as the behavior is consistent with the above expansion.

The placement of `v` inside the while loop is important for how it is captured by any anonymous function occurring in the *embedded_statement*.

For example:

```
int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}

f();
```

If `v` was declared outside of the while loop, it would be shared among all iterations, and its value after the for loop would be the final value, `13`, which is what the invocation of `f` would print. Instead, because each iteration has its own variable `v`, the one captured by `f` in the first iteration will continue to hold the value `7`, which is what will be printed. (Note: earlier versions of C# declared `v` outside of the while loop.)

The body of the finally block is constructed according to the following steps:

- If there is an implicit conversion from `E` to the `System.IDisposable` interface, then
  - If `E` is a non-nullable value type then the finally clause is expanded to the semantic equivalent of:

    ```
    finally {
        ((System.IDisposable)e).Dispose();
    }
    ```

  - Otherwise the finally clause is expanded to the semantic equivalent of:

    ```
    finally {
        if (e != null) ((System.IDisposable)e).Dispose();
    }
    ```

    except that if `E` is a value type, or a type parameter instantiated to a value type, then the cast of `e` to `System.IDisposable` will not cause boxing to occur.
- Otherwise, if `E` is a sealed type, the finally clause is expanded to an empty block:

  ```
  finally {
  }
  ```

- Otherwise, the finally clause is expanded to:

  ```
  finally {
      System.IDisposable d = e as System.IDisposable;
      if (d != null) d.Dispose();
  }
  ```

  The local variable `d` is not visible to or accessible to any user code. In particular, it does not conflict with any other variable whose scope includes the finally block.

The order in which `foreach` traverses the elements of an array, is as follows: For single-dimensional arrays elements are traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

The following example prints out each value in a two-dimensional array, in element order:

```
using System;

class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);

        Console.WriteLine();
    }
}
```

The output produced is as follows:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

In the example

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

the type of `n` is inferred to be `int`, the element type of `numbers`.

Jump statements

Jump statements unconditionally transfer control.

```
jump_statement
    : break_statement
    | continue_statement
    | goto_statement
    | return_statement
    | throw_statement
    ;
```

The location to which a jump statement transfers control is called the ***target*** of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to ***exit*** the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening `try` statements. In the absence of such `try` statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening `try` statements, execution is more complex. If the jump statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the

innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.

In the example

```
using System;

class Test
{
    static void Main() {
        while (true) {
            try {
                try {
                    Console.WriteLine("Before break");
                    break;
                }
                finally {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

the `finally` blocks associated with two `try` statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

```
Before break
Innermost finally block
Outermost finally block
After break
```

The break statement

The `break` statement exits the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement.

```
break_statement
    : 'break' ';'
    ;
```

The target of a `break` statement is the end point of the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement. If a `break` statement is not enclosed by a `switch`, `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `switch`, `while`, `do`, `for`, or `foreach` statements are nested within each other, a `break` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (The goto statement) must be used.

A `break` statement cannot exit a `finally` block (The try statement). When a `break` statement occurs within a `finally` block, the target of the `break` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `break` statement is executed as follows:

- If the `break` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `break` statement.

Because a `break` statement unconditionally transfers control elsewhere, the end point of a `break` statement is never reachable.

The continue statement

The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

```
continue_statement
    : 'continue' ';'
    ;
```

The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (The goto statement) must be used.

A `continue` statement cannot exit a `finally` block (The try statement). When a `continue` statement occurs within a `finally` block, the target of the `continue` statement must be within the same `finally` block; otherwise a compile-time error occurs.

A `continue` statement is executed as follows:

- If the `continue` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `continue` statement.

Because a `continue` statement unconditionally transfers control elsewhere, the end point of a `continue` statement is never reachable.

The goto statement

The `goto` statement transfers control to a statement that is marked by a label.

```
goto_statement
    : 'goto' identifier ';'
    | 'goto' 'case' constant_expression ';'
    | 'goto' 'default' ';'
    ;
```

The target of a `goto` *identifier* statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a `goto` statement to transfer control out of a nested scope, but not into a nested scope. In the example

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                         goto done;

            Console.WriteLine("{0} not found", str);
            continue;
    done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

a `goto` statement is used to transfer control out of a nested scope.

The target of a `goto case` statement is the statement list in the immediately enclosing `switch` statement (The switch statement), 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 (Implicit conversions) 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 (The switch statement), 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 (The try statement). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.

A `goto` statement is executed as follows:

- If the `goto` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `goto` statement.

Because a `goto` statement unconditionally transfers control elsewhere, the end point of a `goto` statement is never reachable.

The return statement

The `return` statement returns control to the current caller of the function in which the `return` statement appears.

```
return_statement
    : 'return' expression? ';'
    ;
```

A `return` statement with no expression can be used only in a function member that does not compute a value, that is, a method with the result type (Method body) `void`, the `set` accessor of a property or indexer, the `add` and `remove` accessors of an event, an instance constructor, a static constructor, or a destructor.

A `return` statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void result type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion (Implicit conversions) must exist from the type of the expression to the return type of the containing function member.

Return statements can also be used in the body of anonymous function expressions (Anonymous function expressions), and participate in determining which conversions exist for those functions.

It is a compile-time error for a `return` statement to appear in a `finally` block (The try statement).

A `return` statement is executed as follows:

- If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function by an implicit conversion. The result of the conversion becomes the result value produced by the function.
- If the `return` statement is enclosed by one or more `try` or `catch` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- If the containing function is not an async function, control is returned to the caller of the containing function along with the result value, if any.
- If the containing function is an async function, control is returned to the current caller, and the result value, if any, is recorded in the return task as described in (Enumerator interfaces).

Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return` statement is never reachable.

The throw statement

The `throw` statement throws an exception.

```
throw_statement
    : 'throw' expression? ';'
    ;
```

A `throw` statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

A `throw` statement with no expression can be used only in a `catch` block, in which case that statement re-throws the exception that is currently being handled by that `catch` block.

Because a `throw` statement unconditionally transfers control elsewhere, the end point of a `throw` statement is never reachable.

When an exception is thrown, control is transferred to the first `catch` clause in an enclosing `try` statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as ***exception propagation***. Propagation of an exception consists of repeatedly evaluating the following steps until a `catch` clause that matches the exception is found. In this description, the ***throw point*** is initially the location at which the exception is thrown.

- In the current function member, each `try` statement that encloses the throw point is examined. For each statement `S`, starting with the innermost `try` statement and ending with the outermost `try` statement, the following steps are evaluated:
  - If the `try` block of `S` encloses the throw point and if S has one or more `catch` clauses, the `catch` clauses are examined in order of appearance to locate a suitable handler for the exception, according to the rules specified in Section The try statement. If a matching `catch` clause is located, the exception propagation is completed by transferring control to the block of that `catch` clause.
  - Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the `finally` block, processing of the current exception is continued.
- If an exception handler was not located in the current function invocation, the function invocation is terminated, and one of the following occurs:
  - If the current function is non-async, the steps above are repeated for the caller of the function with a throw point corresponding to the statement from which the function member was invoked.
  - If the current function is async and task-returning, the exception is recorded in the return task, which is put into a faulted or cancelled state as described in Enumerator interfaces.
  - If the current function is async and void-returning, the synchronization context of the current thread is notified as described in Enumerable interfaces.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

The try statement

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when control leaves the `try` statement.

```
try_statement
    : 'try' block catch_clause+
    | 'try' block finally_clause
    | 'try' block catch_clause+ finally_clause
    ;

catch_clause
    : 'catch' exception_specifier? exception_filter?  block
    ;

exception_specifier
    : '(' type identifier? ')'
    ;

exception_filter
    : 'when' '(' expression ')'
    ;

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 an *exception_specifier*, the type must be `System.Exception`, a type that derives from `System.Exception` or a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class.

When a `catch` clause specifies both an *exception_specifier* with 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` clause. During execution of the *exception_filter* and *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 filter and `catch` block.

A `catch` clause that does not specify an *exception_specifier* is called a general `catch` clause.

Some programming languages may support exceptions that are not representable as an object derived from `System.Exception`, although such exceptions could never be generated by C# code. A general `catch` clause may be used to catch such exceptions. Thus, a general `catch` clause is semantically different from one that specifies the type `System.Exception`, in that the former may also catch exceptions from other languages.

In order to locate a handler for an exception, `catch` clauses are examined in lexical order. If a `catch` clause specifies a type but no exception filter, it is a compile-time error for a later `catch` clause in the same `try` statement to specify a type that is the same as, or is derived from, that type. If a `catch` clause specifies no type and no filter, it must be the last `catch` clause for that `try` statement.

Within a `catch` block, a `throw` statement (The throw statement) with no expression can be used to re-throw the exception that was caught by the `catch` block. Assignments to an exception variable do not alter the exception that is re-thrown.

In the example

```
using System;

class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;                // re-throw
        }
    }

    static void G() {
        throw new Exception("G");
    }

    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

the method `F` catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output produced is:

```
Exception in F: G
Exception in Main: G
```

If the first catch block had thrown `e` instead of rethrowing the current exception, the output produced would be as follows:

```
Exception in F: G
Exception in Main: F
```

It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally` block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement must be within the same `finally` block, or otherwise a compile-time error occurs.

It is a compile-time error for a `return` statement to occur in a `finally` block.

A `try` statement is executed as follows:

- Control is transferred to the `try` block.
- When and if control reaches the end point of the `try` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `try` block:
  - The `catch` clauses, if any, are examined in order of appearance to locate a suitable handler for the

exception. If a `catch` clause does not specify a type, or specifies the exception type or a base type of the exception type:

- If the `catch` clause declares an exception variable, the exception object is assigned to the exception variable.
- If the `catch` clause declares an exception filter, the filter is evaluated. If it evaluates to `false`, the catch clause is not a match, and the search continues through any subsequent `catch` clauses for a suitable handler.
- Otherwise, the `catch` clause is considered a match, and control is transferred to the matching `catch` block.
- When and if control reaches the end point of the `catch` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `catch` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - The exception is propagated to the next enclosing `try` statement.

- If the `try` statement has no `catch` clauses or if no `catch` clause matches the exception:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - The exception is propagated to the next enclosing `try` statement.

The statements of a `finally` block are always executed when control leaves a `try` statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a `break`, `continue`, `goto`, or `return` statement, or as a result of propagating an exception out of the `try` statement.

If an exception is thrown during execution of a `finally` block, and is not caught within the same finally block, the exception is propagated to the next enclosing `try` statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the `throw` statement (The throw statement).

The `try` block of a `try` statement is reachable if the `try` statement is reachable.

A `catch` block of a `try` statement is reachable if the `try` statement is reachable.

The `finally` block of a `try` statement is reachable if the `try` statement is reachable.

The end point of a `try` statement is reachable if both of the following are true:

- The end point of the `try` block is reachable or the end point of at least one `catch` block is reachable.
- If a `finally` block is present, the end point of the `finally` block is reachable.

The checked and unchecked statements

The `checked` and `unchecked` statements are used to control the ***overflow checking context*** for integral-type arithmetic operations and conversions.

```
checked_statement
    : 'checked' block
    ;

unchecked_statement
    : 'unchecked' block
    ;
```

The `checked` statement causes all expressions in the *block* to be evaluated in a checked context, and the `unchecked` statement causes all expressions in the *block* to be evaluated in an unchecked context.

The `checked` and `unchecked` statements are precisely equivalent to the `checked` and `unchecked` operators (The checked and unchecked operators), except that they operate on blocks instead of expressions.

The lock statement

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

```
lock_statement
    : 'lock' '(' expression ')' embedded_statement
    ;
```

The expression of a `lock` statement must denote a value of a type known to be a *reference_type*. No implicit boxing conversion (Boxing conversions) is ever performed for the expression of a `lock` statement, and thus it is a compile-time error for the expression to denote a value of a *value_type*.

A `lock` statement of the form

```
lock (x) ...
```

where `x` is an expression of a *reference_type*, is precisely equivalent to

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

except that `x` is only evaluated once.

While a mutual-exclusion lock is held, code executing in the same execution thread can also obtain and release the lock. However, code executing in other threads is blocked from obtaining the lock until the lock is released.

Locking `System.Type` objects in order to synchronize access to static data is not recommended. Other code might lock on the same type, which can result in deadlock. A better approach is to synchronize access to static data by locking a private static object. For example:

```
class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }

    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```

The using statement

The `using` statement obtains one or more resources, executes a statement, and then disposes of the resource.

```
using_statement
    : 'using' '(' resource_acquisition ')' embedded_statement
    ;

resource_acquisition
    : local_variable_declaration
    | expression
    ;
```

A ***resource*** is a class or struct that implements `System.IDisposable`, which includes a single parameterless method named `Dispose`. Code that is using a resource can call `Dispose` to indicate that the resource is no longer needed. If `Dispose` is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of *resource_acquisition* is *local_variable_declaration* then the type of the *local_variable_declaration* must be either `dynamic` or a type that can be implicitly converted to `System.IDisposable`. If the form of *resource_acquisition* is *expression* then this expression must be implicitly convertible to `System.IDisposable`.

Local variables declared in a *resource_acquisition* are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these local variables (via assignment or the `++` and `--` operators) , take the address of them, or pass them as `ref` or `out` parameters.

A `using` statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a `try` statement that includes a `finally` clause. This `finally` clause disposes of the resource. If a `null` resource is acquired, then no call to `Dispose` is made, and no exception is thrown. If the resource is of type `dynamic` it is dynamically converted through an implicit dynamic conversion (Implicit dynamic conversions) to `IDisposable` during acquisition in order to ensure that the conversion is successful before the usage and disposal.

A `using` statement of the form

```
using (ResourceType resource = expression) statement
```

corresponds to one of three possible expansions. When `ResourceType` is a non-nullable value type, the expansion is

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}
```

Otherwise, when `ResourceType` is a nullable value type or a reference type other than `dynamic`, the expansion is

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}
```

Otherwise, when `ResourceType` is `dynamic`, the expansion is

```
{
    ResourceType resource = expression;
    IDisposable d = (IDisposable)resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
```

In either expansion, the `resource` variable is read-only in the embedded statement, and the `d` variable is inaccessible in, and invisible to, the embedded statement.

An implementation is permitted to implement a given using-statement differently, e.g. for performance reasons, as long as the behavior is consistent with the above expansion.

A `using` statement of the form

```
using (expression) statement
```

has the same three possible expansions. In this case `ResourceType` is implicitly the compile-time type of the `expression`, if it has one. Otherwise the interface `IDisposable` itself is used as the `ResourceType`. The `resource` variable is inaccessible in, and invisible to, the embedded statement.

When a *resource_acquisition* takes the form of a *local_variable_declaration*, it is possible to acquire multiple resources of a given type. A `using` statement of the form

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

is precisely equivalent to a sequence of nested `using` statements:

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
            using (ResourceType rN = eN)
                statement
```

The example below creates a file named `log.txt` and writes two lines of text to the file. The example then opens that same file for reading and copies the contained lines of text to the console.

```
using System;
using System.IO;

class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }

        }
    }
}
```

Since the `TextWriter` and `TextReader` classes implement the `IDisposable` interface, the example can use `using` statements to ensure that the underlying file is properly closed following the write or read operations.

The yield statement

The `yield` statement is used in an iterator block (Blocks) to yield a value to the enumerator object (Enumerator objects) or enumerable object (Enumerable objects) of an iterator or to signal the end of the iteration.

```
yield_statement
    : 'yield' 'return' expression ';'
    | 'yield' 'break' ';'
    ;
```

`yield` is not a reserved word; it has special meaning only when used immediately before a `return` or `break` keyword. In other contexts, `yield` can be used as an identifier.

There are several restrictions on where a `yield` statement can appear, as described in the following.

- It is a compile-time error for a `yield` statement (of either form) to appear outside a *method_body*, *operator_body* or *accessor_body*
- It is a compile-time error for a `yield` statement (of either form) to appear inside an anonymous function.
- It is a compile-time error for a `yield` statement (of either form) to appear in the `finally` clause of a `try` statement.
- It is a compile-time error for a `yield return` statement to appear anywhere in a `try` statement that contains any `catch` clauses.

The following example shows some valid and invalid uses of `yield` statements.

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;        // Ok
        yield break;           // Ok
    }
    finally {
        yield return 2;        // Error, yield in finally
        yield break;           // Error, yield in finally
    }

    try {
        yield return 3;        // Error, yield return in try...catch
        yield break;           // Ok
    }
    catch {
        yield return 4;        // Error, yield return in try...catch
        yield break;           // Ok
    }

    D d = delegate {
        yield return 5;        // Error, yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;            // Error, wrong return type for an iterator block
}
```

An implicit conversion (Implicit conversions) must exist from the type of the expression in the `yield return` statement to the yield type (Yield type) of the iterator.

A `yield return` statement is executed as follows:

- The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
- Execution of the iterator block is suspended. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are not executed at this time.
- The `MoveNext` method of the enumerator object returns `true` to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object's `MoveNext` method resumes execution of the iterator block from where it was last suspended.

A `yield break` statement is executed as follows:

- If the `yield break` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the iterator block. This is either the `MoveNext` method or `Dispose` method of the enumerator object.

Because a `yield break` statement unconditionally transfers control elsewhere, the end point of a `yield break` statement is never reachable.

Namespaces

1/13/2018 • 18 minutes to read • Edit Online

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Using directives (Using directives) are provided to facilitate the use of namespaces.

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
    : extern_alias_directive* using_directive* global_attributes? namespace_member_declaration*
    ;
```

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_directive*s of a compilation unit affect the *global_attributes* and *namespace_member_declaration*s of that compilation unit, but have no effect on other compilation units.

The *global_attributes* (Attributes) of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The *namespace_member_declaration*s of each compilation unit of a program contribute members to a single declaration space called the global namespace. 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.

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 ';'?
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive* namespace_member_declaration* '}'
    ;
```

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_directive*s import the names of other namespaces, types and members, allowing them to be referenced directly instead of through qualified names. The optional *namespace_member_declaration*s contribute members to the declaration space of the namespace. Note that all *using_directive*s 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. For example,

```
namespace N1.N2
{
    class A {}

    class B {}
}
```

is semantically equivalent to

```
namespace N1
{
    namespace N2
    {
        class A {}

        class B {}
    }
}
```

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space (Declarations). 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.

Extern aliases

An *extern_alias_directive* introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and applies also to nested namespaces of the aliased namespace.

```
extern_alias_directive
    : 'extern' 'alias' identifier ';'
    ;
```

The scope of an *extern_alias_directive* extends over the *using_directive*s, *global_attributes* and *namespace_member_declaration*s of its immediately containing compilation unit or namespace body.

Within a compilation unit or namespace body that contains an *extern_alias_directive*, the identifier introduced by the *extern_alias_directive* can be used to reference the aliased namespace. It is a compile-time error for the *identifier* to be the word `global`.

An *extern_alias_directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.

The following program declares and uses two extern aliases, `X` and `Y`, each of which represent the root of a distinct namespace hierarchy:

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

The program declares the existence of the extern aliases `X` and `Y`, but the actual definitions of the aliases are external to the program. The identically named `N.B` classes can now be referenced as `X.N.B` and `Y.N.B`, or, using the namespace alias qualifier, `X::N.B` and `Y::N.B`. An error occurs if a program declares an extern alias for which no external definition is provided.

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 (Namespace and type names) and *simple_name*s (Simple names), 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_directive
    : using_alias_directive
    | using_namespace_directive
    | using_static_directive
    ;
```

A *using_alias_directive* (Using alias directives) introduces an alias for a namespace or type.

A *using_namespace_directive* (Using namespace directives) imports the type members of a namespace.

A *using_static_directive* (Using static directives) imports the nested types and static members of a type.

The scope of a *using_directive* extends over the *namespace_member_declaration*s 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.

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. 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 {}
}
```

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*. 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. 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. 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 {}
}
```

Just like regular members, names introduced by *using_alias_directive*s are hidden by similarly named members in nested scopes. 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`.

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. A *using_alias_directive* may however be affected by *extern_alias_directive*s in the immediately containing compilation unit or namespace body. In the example

```
namespace N1.N2 {}

namespace N3
{
    extern alias E;

    using R1 = E.N;        // OK

    using R2 = N1;         // OK

    using R3 = N1.N2;      // OK

    using R4 = R2.N2;      // Error, R2 unknown
}
```

the last *using_alias_directive* results in a compile-time error because it is not affected by the first *using_alias_directive*. The first *using_alias_directive* does not result in an error since the scope of the extern alias `E` includes the *using_alias_directive*.

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. 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`.

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments. For example:

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;          // Error, cannot name unbound generic type

    using X = N1.A.B;        // Error, cannot name unbound generic type

    using Y = N1.A<int>;     // Ok, can name closed constructed type

    using Z<T> = N1.A<T>;    // Error, using alias cannot have type parameters
}
```

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. 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`.

A *using_namespace_directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. 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.

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. 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`.

When more than one namespace or type imported by *using_namespace_directive*s or *using_static_directive*s in the same compilation unit or namespace body contain types by the same name, references to that name as a *type_name* are considered ambiguous. 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. In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing a *using_alias_directive* that picks a particular `A`. For example:

```
namespace N3
{
    using N1;

    using N2;

    using A = N1.A;

    class B: A {}               // A means N1.A
}
```

Furthermore, when more than one namespace or type imported by *using_namespace_directive*s or *using_static_directive*s in the same compilation unit or namespace body contain types or members by the same name, references to that name as a *simple_name* are considered ambiguous. In the example

```
namespace N1
{
    class A {}
}

class C
{
    public static int A
}

namespace N2
{
    using N1;
    using static C;

    class B
    {
        void M()
        {
            A a = new A();   // Ok, A is unambiguous as a type-name
            A.Equals(2);     // Error, A is ambiguous as a simple-name
    }
}
```

`N1` contains a type member `A`, and `C` contains a static method `A`, and because `N2` imports both, referencing `A` as a *simple_name* is ambiguous and a compile-time error.

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.

Using static directives

A *using_static_directive* imports the nested types and static members contained directly in a type declaration into the immediately enclosing compilation unit or namespace body, enabling the identifier of each member and type to be used without qualification.

```
using_static_directive
    : 'using' 'static' type_name ';'
    ;
```

Within member declarations in a compilation unit or namespace body that contains a *using_static_directive*, the accessible nested types and static members (except extension methods) contained directly in the declaration of the given type can be referenced directly. For example:

```
namespace N1
{
    class A
    {
        public class B{}
        public static B M(){ return new B(); }
    }
}

namespace N2
{
    using static N1.A;
    class C
    {
        void N() { B b = M(); }
    }
}
```

Above, within member declarations in the `N2` namespace, the static members and nested types of `N1.A` are directly available, and thus the method `N` is able to reference both the `B` and `M` members of `N1.A`.

A *using_static_directive* specifically does not import extension methods directly as static methods, but makes them available for extension method invocation (Extension method invocations). In the example

```
namespace N1
{
    static class A
    {
        public static void M(this string s){}
    }
}

namespace N2
{
    using static N1.A;

    class B
    {
        void N()
        {
            M("A");      // Error, M unknown
            "B".M();     // Ok, M known as extension method
            N1.A.M("C"); // Ok, fully qualified
        }
    }
}
```

the *using_static_directive* imports the extension method `M` contained in `N1.A`, but only as an extension method. Thus, the first reference to `M` in the body of `B.N` results in a compile-time error because no members named `M` are in scope.

A *using_static_directive* only imports members and types declared directly in the given type, not members and types declared in base classes.

TODO: Example

Ambiguities between multiple *using_namespace_directives* and *using_static_directives* are discussed in Using namespace directives.

Namespace members

A *namespace_member_declaration* is either a *namespace_declaration* (Namespace declarations) or a *type_declaration* (Type declarations).

```
namespace_member_declaration
    : namespace_declaration
    | type_declaration
    ;
```

A compilation unit or a namespace body can contain *namespace_member_declaration*s, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

Type declarations

A *type_declaration* is a *class_declaration* (Class declarations), a *struct_declaration* (Struct declarations), an *interface_declaration* (Interface declarations), an *enum_declaration* (Enum declarations), or a *delegate_declaration* (Delegate declarations).

```
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 (Nested types).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (Declared accessibility):

- 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.

Namespace alias qualifiers

The ***namespace alias qualifier*** `::` makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The namespace alias qualifier always appears between two identifiers referred to as the left-hand and right-hand identifiers. Unlike the regular `.` qualifier, the left-hand identifier of the `::` qualifier is looked up only as an extern or using alias.

A *qualified_alias_member* is defined as follows:

```
qualified_alias_member
    : identifier '::' identifier type_argument_list?
    ;
```

A *qualified_alias_member* can be used as a *namespace_or_type_name* (Namespace and type names) or as the left operand in a *member_access* (Member access).

A *qualified_alias_member* has one of two forms:

- `N::I<A1, ..., Ak>`, where `N` and `I` represent identifiers, and `<A1, ..., Ak>` is a type argument list. (`K` is always at least one.)
- `N::I`, where `N` and `I` represent identifiers. (In this case, `K` is considered to be zero.)

Using this notation, the meaning of a *qualified_alias_member* is determined as follows:

- If `N` is the identifier `global`, then the global namespace is searched for `I`:
  - If the global namespace contains a namespace named `I` and `K` is zero, then the *qualified_alias_member* refers to that namespace.
  - Otherwise, if the global namespace contains a non-generic type named `I` and `K` is zero, then the *qualified_alias_member* refers to that type.
  - Otherwise, if the global namespace contains a type named `I` that has `K` type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
  - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.
- Otherwise, starting with the namespace declaration (Namespace declarations) immediately containing the *qualified_alias_member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified_alias_member*, the following steps are evaluated until an entity is located:
  - If the namespace declaration or compilation unit contains a *using_alias_directive* that associates `N` with a type, then the *qualified_alias_member* is undefined and a compile-time error occurs.
  - Otherwise, if the namespace declaration or compilation unit contains an *extern_alias_directive* or *using_alias_directive* that associates `N` with a namespace, then:
    - If the namespace associated with `N` contains a namespace named `I` and `K` is zero, then the *qualified_alias_member* refers to that namespace.
    - Otherwise, if the namespace associated with `N` contains a non-generic type named `I` and `K` is zero, then the *qualified_alias_member* refers to that type.
    - Otherwise, if the namespace associated with `N` contains a type named `I` that has `K` type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
    - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.
- Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.

Note that using the namespace alias qualifier with an alias that references a type causes a compile-time error. Also note that if the identifier `N` is `global`, then lookup is performed in the global namespace, even if there is a using alias associating `global` with a type or namespace.

Uniqueness of aliases

Each compilation unit and namespace body has a separate declaration space for extern aliases and using aliases. Thus, while the name of an extern alias or using alias must be unique within the set of extern aliases and using aliases declared in the immediately containing compilation unit or namespace body, an alias is permitted to have the same name as a type or namespace as long as it is used only with the `::` qualifier.

In the example

```
namespace N
{
    public class A {}

    public class B {}
}

namespace N
{
    using A = System.IO;

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

        A::Stream s2;           // Ok
    }
}
```

the name `A` has two possible meanings in the second namespace body because both the class `A` and the using alias `A` are in scope. For this reason, use of `A` in the qualified name `A.Stream` is ambiguous and causes a compile-time error to occur. However, use of `A` with the `::` qualifier is not an error because `A` is looked up only as a namespace alias.

Classes

1/26/2018 • 170 minutes to read • Edit Online

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.

Class declarations

A *class_declaration* is a *type_declaration* (Type declarations) that declares a new class.

```
class_declaration
    : attributes? class_modifier* 'partial'? 'class' identifier type_parameter_list?
      class_base? type_parameter_constraints_clause* class_body ';'?
    ;
```

A *class_declaration* consists of an optional set of *attributes* (Attributes), followed by an optional set of *class_modifier*s (Class modifiers), followed by an optional `partial` modifier, followed by the keyword `class` and an *identifier* that names the class, followed by an optional *type_parameter_list* (Type parameters), followed by an optional *class_base* specification (Class base specification) , followed by an optional set of *type_parameter_constraints_clause*s (Type parameter constraints), followed by a *class_body* (Class body), optionally followed by a semicolon.

A class declaration cannot supply *type_parameter_constraints_clause*s unless it also supplies a *type_parameter_list*.

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

Class modifiers

A *class_declaration* may optionally include a sequence of class modifiers:

```
class_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'abstract'
    | 'sealed'
    | 'static'
    | class_modifier_unsafe
    ;
```

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 The new modifier. 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 (Declared accessibility).

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

Abstract classes

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a 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. 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.

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.

The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

Static classes

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

A static class declaration is subject to the following restrictions:

- A static class may not include a `sealed` or `abstract` modifier. Note, however, that since a static class cannot be

instantiated or derived from, it behaves as if it was both sealed and abstract.

- A static class may not include a *class_base* specification (Class base specification) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type `object`.
- A static class can only contain static members (Static and instance members). Note that constants and nested types are classified as static members.
- A static class cannot have members with `protected` or `protected internal` declared accessibility.

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

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

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

Referencing static class types

A *namespace_or_type_name* (Namespace and type names) is permitted to reference a static class if

- The *namespace_or_type_name* is the `T` in a *namespace_or_type_name* of the form `T.I`, or
- The *namespace_or_type_name* is the `T` in a *typeof_expression* (Argument lists1) of the form `typeof(T)`.

A *primary_expression* (Function members) is permitted to reference a static class if

- The *primary_expression* is the `E` in a *member_access* (Compile-time checking of dynamic overload resolution) of the form `E.I`.

In any other context it is a compile-time error to reference a static class. For example, it is an error for a static class to be used as a base class, a constituent type (Nested types) of a member, a generic type argument, or a type parameter constraint. Likewise, a static class cannot be used in an array type, a pointer type, a `new` expression, a cast expression, an `is` expression, an `as` expression, a `sizeof` expression, or a default value expression.

Partial modifier

The `partial` modifier is used to indicate that this *class_declaration* is a partial type declaration. Multiple partial type declarations with the same name within an enclosing namespace or type declaration combine to form one type declaration, following the rules specified in Partial types.

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

Type parameters

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

```
type_parameter_list
    : '<' type_parameters '>'
    ;

type_parameters
    : attributes? type_parameter
    | type_parameters ',' attributes? type_parameter
    ;

type_parameter
    : identifier
    ;
```

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

Class base specification

A class declaration may include a *class_base* specification, which defines the direct base class of the class and the interfaces (Interfaces) directly implemented by the class.

```
class_base
    : ':' class_type
    | ':' interface_type_list
    | ':' class_type ',' interface_type_list
    ;

interface_type_list
    : interface_type (',' interface_type)*
    ;
```

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

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

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 Inheritance.

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`.

For a constructed class type, if a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type_parameter* in the base class declaration, the corresponding *type_argument* of the constructed type. Given the generic class declarations

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

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

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

The direct base class of a class type must be at least as accessible as the class type itself (Accessibility domains). 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.MulticastDelegate`, `System.Enum`, or `System.ValueType`. Furthermore, a generic class declaration cannot use `System.Attribute` as a direct or indirect base class.

While determining the meaning of the direct base class specification `A` of a class `B`, the direct base class of `B` is temporarily assumed to be `object`. Intuitively this ensures that the meaning of a base class specification cannot recursively depend on itself. The example:

```
class A<T> {
   public class B {}
}

class C : A<C.B> {}
```

is in error since in the base class specification `A<C.B>` the direct base class of `C` is considered to be `object`, and hence (by the rules of Namespace and type names) `C` is not considered to have a member `B`.

The base classes of a class type 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. Referring to the example above, the base classes of `B` are `A` and `object`. In the example

```
class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

the base classes of `D<int>` are `C<int[]>`, `B<IComparable<int[]>>`, `A`, and `object`.

Except for class `object`, every class type 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 reflexive and transitive closure of the ***directly depends on*** relationship.

The example

```
class A: A {}
```

is erroneous because the class depends on itself. Likewise, the example

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Finally, the example

```
class A: B.C {}

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

results in a compile-time error because `A` depends on `B.C` (its direct base class), which depends on `B` (its immediately enclosing class), which circularly depends on `A`.

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

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

```
sealed class A {}

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

class `B` is in error because it attempts to derive from the `sealed` class `A`.

Interface implementations

A *class_base* specification may include a list of interface types, in which case the class is said to directly implement the given interface types. Interface implementations are discussed further in Interface implementations.

Type parameter constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type_parameter_constraints_clauses*.

```
type_parameter_constraints_clause
    : 'where' type_parameter ':' type_parameter_constraints
    ;

type_parameter_constraints
    : primary_constraint
    | secondary_constraints
    | constructor_constraint
    | primary_constraint ',' secondary_constraints
    | primary_constraint ',' constructor_constraint
    | secondary_constraints ',' constructor_constraint
    | primary_constraint ',' secondary_constraints ',' constructor_constraint
    ;

primary_constraint
    : class_type
    | 'class'
    | 'struct'
    ;

secondary_constraints
    : interface_type
    | type_parameter
    | secondary_constraints ',' interface_type
    | secondary_constraints ',' type_parameter
    ;

constructor_constraint
    : 'new' '(' ')'
    ;
```

Each *type_parameter_constraints_clause* consists of the token `where`, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be at most one `where` clause for each type parameter, and the `where` clauses can be listed in any order. Like the `get` and `set` tokens in a property accessor, the `where` token is not a keyword.

The list of constraints given in a `where` clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, `new()`.

A primary constraint can be a class type or the ***reference type constraint*** `class` or the ***value type constraint*** `struct`. A secondary constraint can be a *type_parameter* or *interface_type*.

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

The value type constraint specifies that a type argument used for the type parameter must be a non-nullable value type. All non-nullable struct types, enum types, and type parameters having the value type constraint satisfy this constraint. Note that although classified as a value type, a nullable type (Nullable types) does not satisfy the value type constraint. A type parameter having the value type constraint cannot also have the *constructor_constraint*.

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

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

A *class_type* constraint must satisfy the following rules:

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

A type specified as an *interface_type* constraint must satisfy the following rules:

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

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

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

A type specified as a *type_parameter* constraint must satisfy the following rules:

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

In addition there must be no cycles in the dependency graph of type parameters, where dependency is a transitive relation defined by:

- If a type parameter `T` is used as a constraint for type parameter `S` then `S` ***depends on*** `T`.
- If a type parameter `S` depends on a type parameter `T` and `T` depends on a type parameter `U` then `S` ***depends on*** `U`.

Given this relation, it is a compile-time error for a type parameter to depend on itself (directly or indirectly).

Any constraints must be consistent among dependent type parameters. If type parameter `S` depends on type parameter `T` then:

- `T` must not have the value type constraint. Otherwise, `T` is effectively sealed so `S` would be forced to be the same type as `T`, eliminating the need for two type parameters.
- If `S` has the value type constraint then `T` must not have a *class_type* constraint.
- If `S` has a *class_type* constraint `A` and `T` has a *class_type* constraint `B` then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.
- If `S` also depends on type parameter `U` and `U` has a *class_type* constraint `A` and `T` has a *class_type* constraint `B` then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.

It is valid for `S` to have the value type constraint and `T` to have the reference type constraint. Effectively this limits `T` to the types `System.Object`, `System.ValueType`, `System.Enum`, and any interface type.

If the `where` clause for a type parameter includes a constructor constraint (which has the form `new()`), it is possible to use the `new` operator to create instances of the type (Object creation expressions). Any type argument used for a type parameter with a constructor constraint must have a public parameterless constructor (this constructor implicitly exists for any value type) or be a type parameter having the value type constraint or constructor constraint (see Type parameter constraints for details).

The following are examples of constraints:

```
interface IPrintable
{
    void Print();
}

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

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

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

class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

The following example is in error because it causes a circularity in the dependency graph of the type parameters:

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

The following examples illustrate additional invalid situations:

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

class A {...}

class B {...}

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

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

The ***effective base class*** of a type parameter `T` is defined as follows:

- If `T` has no primary constraints or type parameter constraints, its effective base class is `object`.
- If `T` has the value type constraint, its effective base class is `System.ValueType`.
- If `T` has a *class_type* constraint `C` but no *type_parameter* constraints, its effective base class is `C`.
- If `T` has no *class_type* constraint but has one or more *type_parameter* constraints, its effective base class is the most encompassed type (Lifted conversion operators) in the set of effective base classes of its *type_parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has both a *class_type* constraint and one or more *type_parameter* constraints, its effective base class is the most encompassed type (Lifted conversion operators) in the set consisting of the *class_type* constraint of `T` and the effective base classes of its *type_parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has the reference type constraint but no *class_type* constraints, its effective base class is `object`.

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

These rules ensure that the effective base class is always a *class_type*.

The ***effective interface set*** of a type parameter `T` is defined as follows:

- If `T` has no *secondary_constraints*, its effective interface set is empty.
- If `T` has *interface_type* constraints but no *type_parameter* constraints, its effective interface set is its set of *interface_type* constraints.
- If `T` has no *interface_type* constraints but has *type_parameter* constraints, its effective interface set is the union of the effective interface sets of its *type_parameter* constraints.
- If `T` has both *interface_type* constraints and *type_parameter* constraints, its effective interface set is the union of its set of *interface_type* constraints and the effective interface sets of its *type_parameter* constraints.

A type parameter is ***known to be a reference type*** if it has the reference type constraint or its effective base class is not `object` or `System.ValueType`.

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

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

the methods of `IPrintable` can be invoked directly on `x` because `T` is constrained to always implement `IPrintable`.

Class body

The *class_body* of a class defines the members of that class.

```
class_body
    : '{' class_member_declaration* '}'
    ;
```

Partial types

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

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

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

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

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

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

Attributes

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

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

are equivalent to a declaration such as:

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

Attributes on type parameters combine in a similar fashion.

Modifiers

When a partial type declaration includes an accessibility specification (the `public`, `protected`, `internal`, and `private` modifiers) it must agree with all other parts that include an accessibility specification. If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility (Declared accessibility).

If one or more partial declarations of a nested type include a `new` modifier, no warning is reported if the nested

type hides an inherited member (Hiding through inheritance).

If one or more partial declarations of a class include an `abstract` modifier, the class is considered abstract (Abstract classes). Otherwise, the class is considered non-abstract.

If one or more partial declarations of a class include a `sealed` modifier, the class is considered sealed (Sealed classes). Otherwise, the class is considered unsealed.

Note that a class cannot be both abstract and sealed.

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

Type parameters and constraints

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

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

The example

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

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

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

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

Base class

When a partial class declaration includes a base class specification it must agree with all other parts that include a base class specification. If no part of a partial class includes a base class specification, the base class becomes `System.Object` (Base classes).

Base interfaces

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

In the example

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

partial class C: IC {...}

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

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

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

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

partial class X: IComparable
{
    ...
}
```

Members

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

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

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

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

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

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

Partial methods

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

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

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

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

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

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

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

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

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

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

- It is a compile-time error to create a delegate to method (Delegate creation expressions).
- It is a compile-time error to refer to `M` inside an anonymous function that is converted to an expression tree type (Evaluation of anonymous function conversions to expression tree types).
- Expressions occurring as part of an invocation of `M` do not affect the definite assignment state (Definite assignment), which can potentially lead to compile-time errors.
- `M` cannot be the entry point for an application (Application Startup).

Partial methods are useful for allowing one part of a type declaration to customize the behavior of another part, e.g., one that is generated by a tool. Consider the following partial class declaration:

```
partial class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

If this class is compiled without any other parts, the defining partial method declarations and their invocations will be removed, and the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Assume that another part is given, however, which provides implementing declarations of the partial methods:

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Then the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Name binding

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

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

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

namespace N
{
    using List = Widgets.LinkedList;

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

Class members

The members of a class consist of the members introduced by its *class_member_declaration*s and the members inherited from the direct base class.

```
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 type are divided into the following categories:

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

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

A *class_declaration* creates a new declaration space (Declarations), and the *class_member_declaration*s 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 (Signatures and overloading) of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class, and two constructors declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

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

The instance type

Each class declaration has an associated bound type (Bound and unbound types), the ***instance type***. For a generic class declaration, the instance type is formed by creating a constructed type (Constructed types) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it can only be used where the type parameters are in scope; that is, inside the class declaration. The instance type is the type of `this` for code written inside the class declaration. For non-generic classes, the instance type is simply the declared class. The following shows several class declarations along with their instance types:

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

class D {}                          // instance type: D
```

Members of constructed types

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

For example, given the generic class declaration

```
class Gen<T,U>
{
    public T[,] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

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

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

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

Within instance function members, the type of `this` is the instance type (The instance type) of the containing declaration.

All members of a generic class can use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type (Open and closed types) is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example:

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

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

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

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

Inheritance

A class ***inherits*** the members of its direct base class type. Inheritance means that a class implicitly contains all members of its direct base class type, 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 (Member access). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
- A derived class can ***hide*** (Hiding through inheritance) inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove that member—it merely makes that member inaccessible directly through the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion (Implicit reference conversions) 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.

The inherited member of a constructed class type are the members of the immediate base class type (Base classes), which is found by substituting the type arguments of the constructed type for each occurrence of the corresponding type parameters in the *class_base* specification. These members, in turn, are transformed by substituting, for each *type_parameter* in the member declaration, the corresponding *type_argument* of the *class_base* specification.

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

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

In the above example, the constructed type `D<int>` has a non-inherited member `public int G(string s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class declaration `B`. This inherited member is determined by first determining the base class type `B<int[]>` of `D<int>` by substituting `int` for `T` in the base class specification `B<T[]>`. Then, as a type argument to `B`, `int[]` is substituted for `U` in `public U F(long index)`, yielding the inherited member `public int[] F(long index)`.

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 Hiding through inheritance.

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

Access modifiers

A *class_member_declaration* can have any one of the five possible kinds of declared accessibility (Declared accessibility): `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected internal` combination, it is a compile-time error to specify more than one access modifier. When a *class_member_declaration* does not include any access modifiers, `private` is assumed.

Constituent types

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

Static and instance members

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

When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

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

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a `static`

modifier, it declares an instance member. (An instance member is sometimes called a non-static member.) Instance members have the following characteristics:

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

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* (Simple names) 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* (Member access), instance members must be accessed through instances, and static members must be accessed through types.

Nested types

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

In the example

```
using System;

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

Fully qualified name

The fully qualified name (Fully qualified names) for a nested type is `S.N` where `S` is the fully qualified name of the type in which type `N` is declared.

Declared accessibility

Non-nested types can have `public` or `internal` declared accessibility and have `internal` declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of five forms of declared accessibility (`public`, `protected internal`, `protected`, `internal`, or `private`) and, like other class members, defaults to `private` declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (`public`, `internal`, or `private`) and, like other struct members, defaults to `private` declared accessibility.

The example

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

declares a private nested class `Node`.

Hiding

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

```
using System;

class Base
{
    public static void M() {
        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`.

this access

A nested type and its containing type do not have a special relationship with regard to *this_access* (This access). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the `this` for the instance of the containing type as a constructor argument for the nested type. The following example

```
using System;

class C
{
    int i = 123;

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

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            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.

Access to private and protected members of the containing type

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

```
using System;

class C
{
    private static void F() {
        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.

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

```
using System;

class Base
{
    protected void F() {
        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`.

Nested types in generic classes

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

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

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

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

        Outer<int>.Inner<string>.F(3, "abc");    // This type is different

        Outer.Inner<string>.F(t, "abc");         // Error, Outer needs type arg
    }
}
```

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

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

Reserved member names

To facilitate the underlying C# run-time implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying run-time implementation does not make use of these reservations.

The reserved names do not introduce declarations, thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance (Inheritance), and can be hidden with the `new` modifier (The new modifier).

The reservation of these names serves three purposes:

- To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
- To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
- To help ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

The declaration of a destructor (Destructors) also causes a signature to be reserved (Member names reserved for destructors).

Member names reserved for properties

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

```
T get_P();
void set_P(T value);
```

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

In the example

```
using System;

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

class B: A
{
    new public int get_P() {
        return 456;
    }

    new public void set_P(int value) {
    }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        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
```

Member names reserved for events

For an event `E` (Events) of delegate type `T`, the following signatures are reserved:

```
void add_E(T handler);
void remove_E(T handler);
```

Member names reserved for indexers

For an indexer (Indexers) of type `T` with parameter-list `L`, the following signatures are reserved:

```
T get_Item(L);
void set_Item(L, T value);
```

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

Furthermore the member name `Item` is reserved.

Member names reserved for destructors

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

```
void Finalize();
```

Constants

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

```
constant_declaration
    : attributes? constant_modifier* 'const' type constant_declarators ';'
    ;

constant_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

A *constant_declaration* may include a set of *attributes* (Attributes), a `new` modifier (The new modifier), and a valid combination of the four access modifiers (Access modifiers). The attributes and modifiers apply to all of the members declared by the *constant_declaration*. Even though constants are considered static members, a *constant_declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant_declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant_declarators*, each of which introduces a new member. A *constant_declarator* consists of an *identifier* that names the member, followed by an "`=`" token, followed by a *constant_expression* (Constant expressions) 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 (Implicit conversions).

The *type* of a constant must be at least as accessible as the constant itself (Accessibility constraints).

The value of a constant is obtained in an expression using a *simple_name* (Simple names) or a *member_access* (Member access).

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

As described in Constant expressions, a *constant_expression* is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a *reference_type* other than `string` is to apply the `new` operator, and since the `new` operator is not permitted in a *constant_expression*, the only possible value for constants of *reference_types* other than `string` is `null`.

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

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

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

is equivalent to

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

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

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

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

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

Fields

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

```
field_declaration
    : attributes? field_modifier* type variable_declarators ';'
    ;

field_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'readonly'
    | 'volatile'
    | field_modifier_unsafe
    ;

variable_declarators
    : variable_declarator (',' variable_declarator)*
    ;

variable_declarator
    : identifier ('=' variable_initializer)?
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

A *field_declaration* may include a set of *attributes* (Attributes), a `new` modifier (The new modifier), a valid combination of the four access modifiers (Access modifiers), and a `static` modifier (Static and instance fields). In addition, a *field_declaration* may include a `readonly` modifier (Readonly fields) or a `volatile` modifier (Volatile fields) but not both. The attributes and modifiers apply to all of the members declared by the *field_declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

The *type* of a *field_declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable_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* (Variable initializers) that gives the initial value of that member.

The *type* of a field must be at least as accessible as the field itself (Accessibility constraints).

The value of a field is obtained in an expression using a *simple_name* (Simple names) or a *member_access* (Member access). The value of a non-readonly field is modified using an *assignment* (Assignment operators). The value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators (Postfix increment and decrement operators) and prefix increment and decrement operators (Prefix increment and decrement operators).

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

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

is equivalent to

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

Static and instance fields

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

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

For example:

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

    public C() {
        count++;
    }

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

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

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

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

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

When a field is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static field, `E` must denote a type containing `M`, and if `M` is an instance field, E must denote an instance of a type containing `M`.

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

Readonly fields

When a *field_declaration* includes a `readonly` modifier, the fields introduced by the declaration are ***readonly fields***. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the following contexts:

- In the *variable_declarator* that introduces the field (by including a *variable_initializer* in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a compile-time error.

Using static readonly fields for constants

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-time. In the example

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

    private byte red, green, blue;

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

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

Versioning of constants and static readonly fields

Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time. Consider an application that consists of two separate programs:

```
using System;

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

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

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* (The lock statement). These optimizations can be performed by the compiler, by the run-time system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a ***volatile read***. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a ***volatile write***. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of the following:

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

The example

```
using System;
using System.Threading;

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

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

    static void Main() {
        finished = false;

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

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

produces the output:

```
result = 143
```

In this example, the method `Main` starts a new thread that runs the method `Thread2`. This method stores a value into a non-volatile field called `result`, then stores `true` in the volatile field `finished`. The main thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `finished` has been declared `volatile`, the main thread must read the value `143` from the field `result`. If the field `finished` had not been declared `volatile`, then it would be permissible for the store to `result` to be visible to the main thread after the store to `finished`, and hence for the main thread to read the value `0` from the field `result`. Declaring `finished` as a `volatile` field prevents any such inconsistency.

Field initialization

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

```
using System;

class Test
{
    static bool b;
    int i;

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

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.

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.

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

It is possible for static fields with variable initializers to be observed in their default value state. However, this is strongly discouraged as a matter of style. The example

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        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`.

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 (Static constructors) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. The example

```
using System;

class Test
{
    static void Main() {
        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 (as defined in Static constructors) provide that `B`'s static constructor (and hence `B`'s static field initializers) must run before `A`'s static constructor and field initializers.

Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (Constructor initializers) 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 Instance constructors.

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

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

the variable initializer for `y` results in a compile-time error because it references a member of the instance being created.

Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class.

Methods are declared using *method_declaration*s:

```
method_declaration
    : method_header method_body
    ;

method_header
    : attributes? method_modifier* 'partial'? return_type member_name type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause*
    ;

method_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | 'async'
    | method_modifier_unsafe
    ;

return_type
    : type
    | 'void'
    ;

member_name
    : identifier
    | interface_type '.' identifier
    ;

method_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

A *method_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) 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 (Access modifiers).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: `static`, `virtual`, and `override`.
- The declaration includes at most one of the following modifiers: `new` and `override`.
- If the declaration includes the `abstract` modifier, then the declaration does not include any of the following modifiers: `static`, `virtual`, `sealed` or `extern`.
- If the declaration includes the `private` modifier, then the declaration does not include any of the following modifiers: `virtual`, `override`, or `abstract`.
- If the declaration includes the `sealed` modifier, then the declaration also includes the `override` modifier.
- If the declaration includes the `partial` modifier, then it does not include any of the following modifiers: `new`,

`public`, `protected`, `internal`, `private`, `virtual`, `sealed`, `override`, `abstract`, or `extern`.

A method that has the `async` modifier is an async function and follows the rules described in Async functions.

The *return_type* of a method declaration specifies the type of the value computed and returned by the method. The *return_type* is `void` if the method does not return a value. If the declaration includes the `partial` modifier, then the return type must be `void`.

The *member_name* specifies the name of the method. Unless the method is an explicit interface member implementation (Explicit interface member implementations), the *member_name* is simply an *identifier*. For an explicit interface member implementation, the *member_name* consists of an *interface_type* followed by a "`.`" and an *identifier*.

The optional *type_parameter_list* specifies the type parameters of the method (Type parameters). If a *type_parameter_list* is specified the method is a ***generic method***. If the method has an `extern` modifier, a *type_parameter_list* cannot be specified.

The optional *formal_parameter_list* specifies the parameters of the method (Method parameters).

The optional *type_parameter_constraints_clause*s specify constraints on individual type parameters (Type parameter constraints) and may only be specified if a *type_parameter_list* is also supplied, and the method does not have an `override` modifier.

The *return_type* and each of the types referenced in the *formal_parameter_list* of a method must be at least as accessible as the method itself (Accessibility constraints).

The *method_body* is either a semicolon, a ***statement body*** or an ***expression body***. A statement body consists of a *block*, which specifies the statements to execute when the method is invoked. An expression body consists of `=>` followed by an *expression* and a semicolon, and denotes a single expression to perform when the method is invoked.

For `abstract` and `extern` methods, the *method_body* consists simply of a semicolon. For `partial` methods the *method_body* may consist of either a semicolon, a block body or an expression body. For all other methods, the *method_body* is either a block body or an expression body.

If the *method_body* consists of a semicolon, then the declaration may not include the `async` modifier.

The name, the type parameter list and the formal parameter list of a method define the signature (Signatures and overloading) of the method. Specifically, the signature of a method consists of its name, the number of type parameters and the number, modifiers, and types of its formal parameters. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method.The return type is not part of a method's signature, nor are the names of the type parameters or the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.

The method's *type_parameter*s are in scope throughout the *method_declaration*, and can be used to form types throughout that scope in *return_type*, *method_body*, and *type_parameter_constraints_clause*s but not in *attributes*.

All formal parameters and type parameters must have different names.

Method parameters

The parameters of a method, if any, are declared by the method's *formal_parameter_list*.

```
formal_parameter_list
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : 'ref'
    | 'out'
    | 'this'
    ;

parameter_array
    : attributes? '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* (Attributes), an optional `ref`, `out` or `this` modifier, a *type*, an *identifier* and an optional *default_argument*. Each *fixed_parameter* declares a parameter of the given type with the given name. The `this` modifier designates the method as an extension method and is only allowed on the first parameter of a static method. Extension methods are further described in Extension methods.

A *fixed_parameter* with a *default_argument* is known as an ***optional parameter***, whereas a *fixed_parameter* without a *default_argument* is a ***required parameter***. A required parameter may not appear after an optional parameter in a *formal_parameter_list*.

A `ref` or `out` parameter cannot have a *default_argument*. The *expression* in a *default_argument* must be one of the following:

- a *constant_expression*
- an expression of the form `new S()` where `S` is a value type
- an expression of the form `default(S)` where `S` is a value type

The *expression* must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

If optional parameters occur in an implementing partial method declaration (Partial methods) , an explicit interface member implementation (Explicit interface member implementations) or in a single-parameter indexer declaration (Indexers) the compiler should give a warning, since these members can never be invoked in a way that permits arguments to be omitted.

A *parameter_array* consists of an optional set of *attributes* (Attributes), 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 (Array types). 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 Parameter arrays.

A *parameter_array* may occur after an optional parameter, but cannot have a default value -- the omission of arguments for a *parameter_array* would instead result in the creation of an empty array.

The following example illustrates different kinds of parameters:

```
public void M(
    ref int      i,
    decimal      d,
    bool         b = false,
    bool?        n = false,
    string       s = "Hello",
    object       o = null,
    T            t = default(T),
    params int[] a
) { }
```

In the *formal_parameter_list* for `M`, `i` is a required ref parameter, `d` is a required value parameter, `b`, `s`, `o` and `t` are optional value parameters and `a` is a parameter array.

A method declaration creates a separate declaration space for parameters, type parameters and local variables. Names are introduced into this declaration space by the type parameter list and the formal parameter list of the method and by local variable declarations in the *block* of the method. It is an error for two members of a method declaration space to have the same name. It is an error for the method declaration space and the local variable declaration space of a nested declaration space to contain elements with the same name.

A method invocation (Method invocations) 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 (Simple names).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the `ref` modifier.
- Output parameters, which are declared with the `out` modifier.
- Parameter arrays, which are declared with the `params` modifier.

As described in Signatures and overloading, the `ref` and `out` modifiers are part of a method's signature, but the `params` modifier is not.

Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression that is implicitly convertible (Implicit conversions) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter—they have no effect on the actual argument given in the method invocation.

Reference parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a *variable_reference* (Precise rules for determining definite assignment)

of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

A method declared as an iterator (Iterators) cannot have reference parameters.

The example

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        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`.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the example

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

Output parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must

consist of the keyword `out` followed by a *variable_reference* (Precise rules for determining definite assignment) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

A method declared as a partial method (Partial methods) or an iterator (Iterators) cannot have output parameters.

Output parameters are typically used in methods that produce multiple return values. For example:

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        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.

Parameter arrays

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. It is not possible to combine the `params` modifier with the modifiers `ref` and `out`.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression that is implicitly convertible (Implicit conversions) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression that is implicitly convertible (Implicit conversions) 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 (Value parameters) of the same type.

The example

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        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[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form (Applicable function member). The expanded form of a method is available only if the normal form of the method is not applicable and only if an applicable method with the same signature as the expanded form is not already declared in the same type.

The example

```
using System;

class Test
{
    static void F(params object[] a) {
        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.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expended form for a single `object` parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

The example

```
using System;

class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        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[]`).

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` (This access).

When a method is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static method, `E` must denote a type containing `M`, and if `M` is an instance method, `E` must denote an instance of a type containing `M`.

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

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 (Override methods).

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 Method invocations.
- Then, if `M` is a non-virtual method, `M` is invoked.
- Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

For every virtual method declared in or inherited by a class, there exists a ***most derived implementation*** of the method with respect to that class. The most derived implementation of a virtual method `M` with respect to a class `R` is determined as follows:

- If `R` contains the introducing `virtual` declaration of `M`, then this is the most derived implementation of `M`.
- Otherwise, if `R` contains an `override` of `M`, then this is the most derived implementation of `M`.
- Otherwise, the most derived implementation of `M` with respect to `R` is the same as the most derived implementation of `M` with respect to the direct base class of `R`.

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.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. In the example

```
using System;

class A
{
    public virtual void F() { 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.

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 type of `C`, starting with the direct base class type of `C` and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located which has the same signature as `M` after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is `public`, if it is `protected`, if it is `protected internal`, or if it is `internal` and declared in the same program as `C`.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- There is exactly one such overridden base method. This restriction has effect only if the base class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override method and the overridden base method have the same return type.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method. However, if the overridden base method is protected internal and it is declared in a different assembly than the assembly containing the override method then the override method's declared accessibility must be protected.
- The override declaration does not specify type-parameter-constraints-clauses. Instead the constraints are inherited from the overridden base method. Note that constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not legal when explicitly specified, such as value types or sealed types.

The following example demonstrates how the overriding rules work for generic classes:

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}            // Ok
    public override C<string> G() {...}         // Ok
    public override void H(C<T> x) {...}        // Error, should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}                 // Ok
    public override C<U> G() {...}              // Ok
    public override void H(C<T> x) {...}        // Error, should be C<U>
}
```

An override declaration can access the overridden base method using a *base_access* (Base access). 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`.

Only by including an `override` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {}        // Warning, hiding inherited F()
}
```

the `F` method in `B` does not include an `override` modifier and therefore does not override the `F` method in `A`. Rather, the `F` method in `B` hides the method in `A`, and a warning is reported because the declaration does not include a `new` modifier.

In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}        // Hides A.F within body of B
}

class C: B
{
    public override void F() {}    // Ok, overrides A.F
}
```

the `F` method in `B` hides the virtual `F` method inherited from `A`. Since the new `F` in `B` has private access, its scope only includes the class body of `B` and does not extend to `C`. Therefore, the declaration of `F` in `C` is permitted to override the `F` inherited from `A`.

Sealed methods

When an instance method declaration includes a `sealed` modifier, that method is said to be a ***sealed method***. If an instance method declaration includes the `sealed` modifier, it must also include the `override` modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method.

In 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`.

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 (Abstract classes).

In the example

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

the `Shape` class defines the abstract notion of a geometrical shape object that can paint itself. The `Paint` method is abstract because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are non-abstract, they are required to override the `Paint` method and provide an actual implementation.

It is a compile-time error for a *base_access* (Base access) to reference an abstract method. In the example

```
abstract class A
{
    public abstract void F();
}

class B: A
{
    public override void F() {
        base.F();                       // Error, base.F is abstract
    }
}
```

a compile-time error is reported for the `base.F()` invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides the abstract method to provide its own implementation.

External methods

When a method declaration includes an `extern` modifier, that method is said to be an ***external method***. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method_body* of an external method simply consists of a semicolon. An external method may not be generic.

The `extern` modifier is typically used in conjunction with a `DllImport` attribute (Interoperation with COM and Win32 components), allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a `DllImport` attribute, the method declaration must also include a `static` modifier. This example demonstrates the use of the `extern` modifier and the `DllImport` attribute:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

Partial methods (recap)

When a method declaration includes a `partial` modifier, that method is said to be a ***partial method***. Partial methods can only be declared as members of partial types (Partial types), and are subject to a number of

restrictions. Partial methods are further described in Partial methods.

Extension methods

When the first parameter of a method includes the `this` modifier, that method is said to be an ***extension method***. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than `this`, and the parameter type cannot be a pointer type.

The following is an example of a static class that declares two extension methods:

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax (Extension method invocations), using the receiver expression as the first argument.

The following program uses the extension methods declared above:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

The `Slice` method is available on the `string[]`, and the `ToInt32` method is available on `string`, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

Method body

The *method_body* of a method declaration consists of either a block body, an expression body or a semicolon.

The ***result type*** of a method is `void` if the return type is `void`, or if the method is async and the return type is `System.Threading.Tasks.Task`. Otherwise, the result type of a non-async method is its return type, and the result

type of an async method with return type `System.Threading.Tasks.Task<T>` is `T`.

When a method has a `void` result type and a block body, `return` statements (The return statement) in the block are not permitted to specify an expression. If execution of the block of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its current caller.

When a method has a `void` result and an expression body, the expression `E` must be a *statement_expression*, and the body is exactly equivalent to a block body of the form `{ E; }`.

When a method has a non-void result type and a block body, each `return` statement in the block must specify an expression that is implicitly convertible to the result type. The endpoint of a block body of a value-returning method must not be reachable. In other words, in a value-returning method with a block body, control is not permitted to flow off the end of the method body.

When a method has a non-void result type and an expression body, the expression must be implicitly convertible to the result type, and the body is exactly equivalent to a block body of the form `{ return E; }`.

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;
        }
    }

    public int I(bool b) => b ? 1 : 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. The `I` method is correct, because its body is equivalent to a statement block with just a single return statement in it.

Method overloading

The method overload resolution rules are described in Type inference.

Properties

A ***property*** is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes; furthermore, they permit such attributes to be computed.

Properties are declared using *property_declaration*s:

```
property_declaration
    : attributes? property_modifier* type member_name property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | property_modifier_unsafe
    ;

property_body
    : '{' accessor_declarations '}' property_initializer?
    | '=>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;
```

A *property_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Property declarations are subject to the same rules as method declarations (Methods) 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 (Explicit interface member implementations), 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 (Accessibility constraints).

A *property_body* may either consist of an ***accessor body*** or an ***expression body***. In an accessor body, *accessor_declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (Accessors) of the property. The accessors specify the executable statements associated with reading and writing the property.

An expression body consisting of `=>` followed by an *expression* `E` and a semicolon is exactly equivalent to the statement body `{ get { return E; } }`, and can therefore only be used to specify getter-only properties where the result of the getter is given by a single expression.

A *property_initializer* may only be given for an automatically implemented property (Automatically implemented properties), and causes the initialization of the underlying field of such properties with the value given by the *expression*.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a `ref` or `out` argument.

When a property declaration includes an `extern` modifier, the property is said to be an ***external property***. Because an external property declaration provides no actual implementation, each of its *accessor_declarations*

consists of a semicolon.

Static and instance properties

When a property declaration includes a `static` modifier, the property is said to be a ***static property***. When no `static` modifier is present, the property is said to be an ***instance property***.

A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as `this` (This access) in the accessors of that property.

When a property is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static property, `E` must denote a type containing `M`, and if `M` is an instance property, E must denote an instance of a type containing `M`.

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

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?
    | set_accessor_declaration get_accessor_declaration?
    ;

get_accessor_declaration
    : attributes? accessor_modifier? 'get' accessor_body
    ;

set_accessor_declaration
    : attributes? accessor_modifier? 'set' accessor_body
    ;

accessor_modifier
    : 'protected'
    | 'internal'
    | 'private'
    | 'protected' 'internal'
    | 'internal' 'protected'
    ;

accessor_body
    : block
    | ';'
    ;
```

The accessor declarations consist of a *get_accessor_declaration*, a *set_accessor_declaration*, or both. Each accessor declaration consists of the token `get` or `set` followed by an optional *accessor_modifier* and an *accessor_body*.

The use of *accessor_modifiers* is governed by the following restrictions:

- An *accessor_modifier* may not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no `override` modifier, an *accessor_modifier* is permitted only if the property or indexer has both a `get` and `set` accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an `override` modifier, an accessor must match the *accessor_modifier*, if any, of the accessor being overridden.
- The *accessor_modifier* must declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:

- If the property or indexer has a declared accessibility of `public`, the *accessor_modifier* may be either `protected internal`, `internal`, `protected`, or `private`.
- If the property or indexer has a declared accessibility of `protected internal`, the *accessor_modifier* may be either `internal`, `protected`, or `private`.
- If the property or indexer has a declared accessibility of `internal` or `protected`, the *accessor_modifier* must be `private`.
- If the property or indexer has a declared accessibility of `private`, no *accessor_modifier* may be used.

For `abstract` and `extern` properties, the *accessor_body* for each accessor specified is simply a semicolon. A non-abstract, non-extern property may have each *accessor_body* be a semicolon, in which case it is an ***automatically implemented property*** (Automatically implemented properties). An automatically implemented property must have at least a get accessor. For the accessors of any other non-abstract, non-extern property, the *accessor_body* is a *block* which specifies the statements to be executed when the corresponding accessor is invoked.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property (Values of expressions). The body of a `get` accessor must conform to the rules for value-returning methods described in Method body. 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 (Assignment operators), or as the operand of `++` or `--` (Postfix increment and decrement operators, Prefix increment and decrement operators), 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 (Simple assignment). The body of a `set` accessor must conform to the rules for `void` methods described in Method body. In particular, `return` statements in the `set` accessor body are not permitted to specify an expression. Since a `set` accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declaration in a `set` accessor to have that name.

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

- A property that includes both a `get` accessor and a `set` accessor is said to be a ***read-write*** property.
- A property that has only a `get` accessor is said to be a ***read-only*** property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a ***write-only*** property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns the string stored in the private `caption` field. The `set` accessor checks if the new value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The `get` accessor simply returns a value stored in a private field, and the `set` accessor modifies that private field and then performs any additional actions required to fully update the state of the object.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```
Button okButton = new Button();
okButton.Caption = "OK";            // Invokes set accessor
string s = okButton.Caption;        // Invokes get accessor
```

Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by referencing the property in an expression.

The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. As such, it is not possible for the two accessors of a read-write property to have different accessibility. The example

```
class A
{
    private string name;

    public string Name {                // Error, duplicate member name
        get { return name; }
    }

    public string Name {                // Error, duplicate member name
        set { name = value; }
    }
}
```

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. In the example

```
class A
{
    public int P {
        set {...}
    }
}

class B: A
{
    new public int P {
        get {...}
    }
}
```

the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing. Thus, in the statements

```
B b = new B();
b.P = 1;          // Error, B.P is read-only
((A)b).P = 1;     // Ok, reference to A.P
```

the assignment to `b.P` causes a compile-time error to be reported, since the read-only `P` property in `B` hides the write-only `P` property in `A`. Note, however, that a cast can be used to access the hidden `P` property.

Unlike public fields, properties provide a separation between an object's internal state and its public interface. Consider the example:

```
class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Here, the `Label` class uses two `int` fields, `x` and `y`, to store its location. The location is publicly exposed both as an `X` and a `Y` property and as a `Location` property of type `Point`. If, in a future version of `Label`, it becomes more convenient to store the location as a `Point` internally, the change can be made without affecting the public interface of the class:

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Had `x` and `y` instead been `public readonly` fields, it would have been impossible to make such a change to the `Label` class.

Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as ***inlining***, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for `get` accessors to have observable side-effects. In the example

```
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```

the value of the `Next` property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side-effect, and the property should be implemented as a method instead.

The "no side-effects" convention for `get` accessors doesn't mean that `get` accessors should always be written to simply return values stored in fields. Indeed, `get` accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed `get` accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }

    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}
```

The `Console` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `Console` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in

```
Console.Out.WriteLine("hello, world");
```

the underlying `TextWriter` for the output device is created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

Automatically implemented properties

An automatically implemented property (or ***auto-property*** for short), is a non-abstract non-extern property with semicolon-only accessor bodies. Auto-properties must have a get accessor and can optionally have a set accessor.

When a property is specified as an automatically implemented property, a hidden backing field is automatically available for the property, and the accessors are implemented to read from and write to that backing field. If the auto-property has no set accessor, the backing field is considered `readonly` (Readonly fields). Just like a `readonly` field, a getter-only auto-property can also be assigned to in the body of a constructor of the enclosing class. Such an assignment assigns directly to the readonly backing field of the property.

An auto-property may optionally have a *property_initializer*, which is applied directly to the backing field as a *variable_initializer* (Variable initializers).

The following example:

```
public class Point {
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

is equivalent to the following declaration:

```
public class Point {
    private int __x = 0;
    private int __y = 0;
    public int X { get { return __x; } set { __x = value; } }
    public int Y { get { return __y; } set { __y = value; } }
}
```

The following example:

```
public class ReadOnlyPoint
{
    public int X { get; }
    public int Y { get; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

is equivalent to the following declaration:

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

Notice that the assignments to the readonly field are legal, because they occur within the constructor.

Accessibility

If an accessor has an *accessor_modifier*, the accessibility domain (Accessibility domains) of the accessor is determined using the declared accessibility of the *accessor_modifier*. If an accessor does not have an *accessor_modifier*, the accessibility domain of the accessor is determined from the declared accessibility of the property or indexer.

The presence of an *accessor_modifier* never affects member lookup (Operators) or overload resolution (Overload resolution). The modifiers on the property or indexer always determine which property or indexer is bound to, regardless of the context of the access.

Once a particular property or indexer has been selected, the accessibility domains of the specific accessors involved are used to determine if that usage is valid:

- If the usage is as a value (Values of expressions), the `get` accessor must exist and be accessible.
- If the usage is as the target of a simple assignment (Simple assignment), the `set` accessor must exist and be accessible.
- If the usage is as the target of compound assignment (Compound assignment), or as the target of the `++` or `--` operators (Function members.9, Invocation expressions), both the `get` accessors and the `set` accessor must exist and be accessible.

In the following example, the property `A.Text` is hidden by the property `B.Text`, even in contexts where only the `set` accessor is called. In contrast, the property `B.Count` is not accessible to class `M`, so the accessible property `A.Count` is used instead.

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
    }
}

class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;             // Calls A.Count set accessor
        int i = b.Count;          // Calls A.Count get accessor
        b.Text = "howdy";         // Error, B.Text set accessor not accessible
        string s = b.Text;        // Calls B.Text get accessor
    }
}
```

An accessor that is used to implement an interface may not have an *accessor_modifier*. If only one accessor is used to implement an interface, the other accessor may be declared with an *accessor_modifier*:

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
    public Prop {
        get { return "April"; }       // Must not have a modifier here
        internal set {...}            // Ok, because I.Prop has no set accessor
    }
}
```

Virtual, sealed, override, and abstract property 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 (Abstract classes).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 Virtual methods, Override methods, Sealed methods, and Abstract methods apply as if accessors were methods of a corresponding form:

- A `get` accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
- A `set` accessor corresponds to a method with a single value parameter of the property type, a `void` return type, and the same modifiers as the containing property.

In the example

```
abstract class A
{
    int y;

    public virtual int X {
        get { return 0; }
    }

    public virtual int Y {
        get { return y; }
        set { y = value; }
    }

    public abstract int Z { get; set; }
}
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an abstract read-write property. Because `Z` is abstract, the containing class `A` must also be declared abstract.

A class that derives from `A` is show below:

```
class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}
```

Here, the declarations of `X`, `Y`, and `Z` are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The `get` accessor of `X` and the `set` accessor of `Y` use the `base` keyword to access the inherited accessors. The declaration of `Z` overrides both abstract accessors—thus, there are no outstanding abstract function members in `B`, and `B` is permitted to be a non-abstract class.

When a property is declared as an `override`, any overridden accessors must be accessible to the overriding code. In addition, the declared accessibility of both the property or indexer itself, and of the accessors, must match that of the overridden member and accessors. For example:

```
public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}            // Must specify protected here
        get {...}                      // Must not have a modifier here
    }
}
```

Events

An ***event*** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying ***event handlers***.

Events are declared using *event_declarations*:

```
event_declaration
    : attributes? event_modifier* 'event' type variable_declarators ';'
    | attributes? event_modifier* 'event' type member_name '{' event_accessor_declarations '}'
    ;

event_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | event_modifier_unsafe
    ;

event_accessor_declarations
    : add_accessor_declaration remove_accessor_declaration
    | remove_accessor_declaration add_accessor_declaration
    ;

add_accessor_declaration
    : attributes? 'add' block
    ;

remove_accessor_declaration
    : attributes? 'remove' block
    ;
```

An *event_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Event declarations are subject to the same rules as method declarations (Methods) with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate_type* (Reference types), and that *delegate_type* must be at least as accessible as the event itself (Accessibility constraints).

An event declaration may include *event_accessor_declarations*. However, if it does not, for non-extern, non-abstract events, the compiler supplies them automatically (Field-like events); for extern events, the accessors are provided externally.

An event declaration that omits *event_accessor_declarations* defines one or more events—one for each of the *variable_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_declarations*.

When an event declaration includes an `extern` modifier, the event is said to be an ***external event***. Because an external event declaration provides no actual implementation, it is an error for it to include both the `extern` modifier and *event_accessor_declarations*.

It is a compile-time error for a *variable_declarator* of an event declaration with an `abstract` or `external` modifier to include a *variable_initializer*.

An event can be used as the left-hand operand of the `+=` and `-=` operators (Event assignment). These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since `+=` and `-=` are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

In an operation of the form `x += y` or `x -= y`, when `x` is an event and the reference takes place outside the type that contains the declaration of `x`, the result of the operation has type `void` (as opposed to having the type of `x`, with the value of `x` after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the `Button` class:

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

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events.

Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly include *event_accessor_declarations*. Such an event can be used in any context that permits a field. The field contains a delegate (Delegates) which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `null`.

In the example

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

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}
```

`Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The `OnClick` method in the `Button` class "raises" the `Click` event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators, as in

```
b.Click += new EventHandler(...);
```

which appends a delegate to the invocation list of the `Click` event, and

```
b.Click -= new EventHandler(...);
```

which removes a delegate from the invocation list of the `Click` event.

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock (The lock statement) on the containing object for an instance event, or the type object (Anonymous object creation expressions) for a static event.

Thus, an instance event declaration of the form:

```
class X
{
    public event D Ev;
}
```

will be compiled to something equivalent to:

```
class X
{
    private D __Ev;  // field to hold the delegate

    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }

        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}
```

Within the class `X`, references to `Ev` on the left-hand side of the `+=` and `-=` operators cause the add and remove accessors to be invoked. All other references to `Ev` are compiled to reference the hidden field `__Ev` instead (Member access). The name "`__Ev`" is arbitrary; the hidden field could have any name or no name at all.

Event accessors

Event declarations typically omit *event_accessor_declarations*, as in the `Button` example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include *event_accessor_declarations* and use a private mechanism for storing the list of event handlers.

The *event_accessor_declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add_accessor_declaration* and a *remove_accessor_declaration*. Each accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an *add_accessor_declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove_accessor_declaration* specifies the statements to execute when an event handler is removed.

Each *add_accessor_declaration* and *remove_accessor_declaration* corresponds to a method with a single value parameter of the event type and a `void` return type. The implicit parameter of an event accessor is named `value`. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is `+=` then the add accessor is used, and if the assignment operator is `-=` then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add_accessor_declaration* or a *remove_accessor_declaration* must conform to the rules for `void` methods described in Method body. In particular, `return` statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

In the example

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }

    // Invoke the MouseUp event
    protected void OnMouseUp(MouseEventArgs args) {
        MouseEventHandler handler;
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        if (handler != null)
            handler(this, args);
    }
}
```

the `Control` class implements an internal storage mechanism for events. The `AddEventHandler` method associates a delegate value with a key, the `GetEventHandler` method returns the delegate currently associated with a key, and the `RemoveEventHandler` method removes a delegate as an event handler for the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for associating a `null` delegate value with a key, and thus unhandled events consume no storage.

Static and instance events

When an event declaration includes a `static` modifier, the event is said to be a ***static event***. When no `static` modifier is present, the event is said to be an ***instance event***.

A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as `this` (This access) in the accessors of that event.

When an event is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static event, `E` must denote a type containing `M`, and if `M` is an instance event, E must denote an instance of a type containing `M`.

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

Virtual, sealed, override, and abstract event accessors

A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier applies to both accessors of an event.

An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an abstract event declaration provides no actual implementation, it cannot provide brace-delimited *event_accessor_declarations*.

An event declaration that includes both the `abstract` and `override` modifiers specifies that the event is abstract and overrides a base event. The accessors of such an event are also abstract.

Abstract event declarations are only permitted in abstract classes (Abstract classes).

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 Virtual methods, Override methods, Sealed methods, and Abstract methods apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a `void` return type, and the same modifiers as the containing event.

Indexers

An ***indexer*** is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer_declaration*s:

```
indexer_declaration
    : attributes? indexer_modifier* indexer_declarator indexer_body
    ;

indexer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | indexer_modifier_unsafe
    ;

indexer_declarator
    : type 'this' '[' formal_parameter_list ']'
    | type interface_type '.' 'this' '[' formal_parameter_list ']'
    ;

indexer_body
    : '{' accessor_declarations '}'
    | '=>' expression ';'
    ;
```

An *indexer_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Indexer declarations are subject to the same rules as method declarations (Methods) 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 (Method parameters), 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 (Accessibility constraints).

An *indexer_body* may either consist of an ***accessor body*** or an ***expression body***. In an accessor body, *accessor_declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (Accessors) of the property. The accessors specify the executable statements associated with reading and writing the property.

An expression body consisting of "`=>`" followed by an expression `E` and a semicolon is exactly equivalent to the statement body `{ get { return E; } }`, and can therefore only be used to specify getter-only indexers where the result of the getter is given by a single expression.

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 (Signatures and overloading) 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* (Simple names) or a *member_access* (Member access), whereas an indexer element is accessed through an *element_access* (Indexer access).
- 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.
- There is no concept of an "automatically implemented indexer". It is an error to have a non-abstract, non-external indexer with semicolon accessors.

Aside from these differences, all rules defined in Accessors and Automatically implemented properties apply to indexer accessors as well as to property accessors.

When an indexer declaration includes an `extern` modifier, the indexer is said to be an ***external indexer***. Because an external indexer declaration provides no actual implementation, each of its *accessor_declarations* consists of a semicolon.

The example below declares a `BitArray` class that implements an indexer for accessing the individual bits in the bit array.

```
using System;

class BitArray
{
    int[] bits;
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```

An instance of the `BitArray` class consumes substantially less memory than a corresponding `bool[]` (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same operations as a `bool[]`.

The following `CountPrimes` class uses a `BitArray` and the classical "sieve" algorithm to compute the number of primes between 1 and a given maximum:

```
class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        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[]`.

The following example shows a 26 * 10 grid class that has an indexer with two parameters. The first parameter is required to be an upper- or lowercase letter in the range A-Z, and the second is required to be an integer in the range 0-9.

```
using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}
```

Indexer overloading

The indexer overload resolution rules are described in Type inference.

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? operator_modifier+ operator_declarator operator_body
    ;

operator_modifier
    : 'public'
    | 'static'
    | 'extern'
    | operator_modifier_unsafe
    ;

operator_declarator
    : unary_operator_declarator
    | binary_operator_declarator
    | conversion_operator_declarator
    ;

unary_operator_declarator
    : type 'operator' overloadable_unary_operator '(' type identifier ')'
    ;

overloadable_unary_operator
    : '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
    ;

binary_operator_declarator
    : type 'operator' overloadable_binary_operator '(' type identifier ',' type identifier ')'
    ;

overloadable_binary_operator
    : '+'   | '-'   | '*'   | '/'   | '%'   | '&'   | '|'   | '^'   | '<<'
    | 'right_shift' | '=='  | '!='  | '>'   | '<'   | '>='  | '<='
    ;

conversion_operator_declarator
    : 'implicit' 'operator' type '(' type identifier ')'
    | 'explicit' 'operator' type '(' type identifier ')'
    ;

operator_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

There are three categories of overloadable operators: Unary operators (Unary operators), binary operators (Binary operators), and conversion operators (Conversion operators).

The *operator_body* is either a semicolon, a ***statement body*** or an ***expression body***. A statement body consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* must conform to the rules for value-returning methods described in Method body. An expression body consists of `=>` followed by an expression and a semicolon, and denotes a single expression to perform when the operator is invoked.

For `extern` operators, the *operator_body* consists simply of a semicolon. For all other operators, the *operator_body* is either a block body or an expression body.

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 (Value parameters). It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator (Unary operators, Binary operators, Conversion operators) 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 (Accessibility constraints).
- 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 Operators.

Additional information on conversion operators can be found in User-defined conversions.

Unary operators

The following rules apply to unary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type `T` or `T?` and can return any type.
- A unary `++` or `--` operator must take a single parameter of type `T` or `T?` and must return that same type or a type derived from it.
- A unary `true` or `false` operator must take a single parameter of type `T` or `T?` and must return type `bool`.

The signature of a unary operator consists of the operator token (`+`, `-`, `!`, `~`, `++`, `--`, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The `true` and `false` unary operators require pair-wise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in User-defined conditional logical operators and Boolean expressions.

The following example shows an implementation and subsequent usage of `operator ++` for an integer vector class:

```
public class IntVector
{
    public IntVector(int length) {...}

    public int Length {...}                // read-only property

    public int this[int index] {...}       // read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);    // vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;    // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;    // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (Postfix increment and decrement operators), and the prefix increment and decrement operators (Prefix increment and decrement operators). Unlike in C++, this method need not modify the value of its operand directly. In fact, modifying the operand value would violate the standard semantics of the postfix increment operator.

Binary operators

The following rules apply to binary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A binary non-shift operator must take two parameters, at least one of which must have type `T` or `T?`, and can return any type.
- A binary `<<` or `>>` operator must take two parameters, the first of which must have type `T` or `T?` and the second of which must have type `int` or `int?`, and can return any type.

The signature of a binary operator consists of the operator token (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, or `<=`) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

- `operator ==` and `operator !=`
- `operator >` and `operator <`
- `operator >=` and `operator <=`

Conversion operators

A conversion operator declaration introduces a ***user-defined conversion*** (User-defined conversions) 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 Implicit conversions.

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 Explicit conversions.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let `S0` and `T0` refer to their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S0` and `T0` are different types.
- Either `S0` or `T0` is the class or struct type in which the operator declaration takes place.
- Neither `S0` nor `T0` is an *interface_type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

For the purposes of these rules, any type parameters associated with `S` or `T` are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

In the example

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}      // Ok
    public static implicit operator C<string>(D<T> value) {...}   // Ok
    public static implicit operator C<T>(D<T> value) {...}        // Error
}
```

the first two operator declarations are permitted because, for the purposes of Indexers.3, `T` and `int` and `string` respectively are considered unique types with no relationship. However, the third operator is an error because `C<T>` is the base class of `D<T>`.

From the second rule it follows that a conversion operator must convert either to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type `C` to define a conversion from `C` to `int` and from `int` to `C`, but not from `int` to `bool`.

It is not possible to directly redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to `object` because implicit and explicit conversions already exist between `object` and all other types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since a conversion would then already exist.

However, it is possible to declare operators on generic types that, for particular type arguments, specify conversions that already exist as pre-defined conversions. In the example

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

when type `object` is specified as a type argument for `T`, the second operator declares a conversion that already exists (an implicit, and therefore also an explicit, conversion exists from any type to type `object`).

In cases where a pre-defined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

- If a pre-defined implicit conversion (Implicit conversions) exists from type `S` to type `T`, all user-defined conversions (implicit or explicit) from `S` to `T` are ignored.
- If a pre-defined explicit conversion (Explicit conversions) exists from type `S` to type `T`, any user-defined explicit conversions from `S` to `T` are ignored. Furthermore:

If `T` is an interface type, user-defined implicit conversions from `S` to `T` are ignored.

Otherwise, user-defined implicit conversions from `S` to `T` are still considered.

For all types but `object`, the operators declared by the `Convertible<T>` type above do not conflict with pre-defined conversions. For example:

```
void F(int i, Convertible<int> n) {
    i = n;                          // Error
    i = (int)n;                     // User-defined explicit conversion
    n = i;                          // User-defined implicit conversion
    n = (Convertible<int>)i;        // User-defined implicit conversion
}
```

However, for type `object`, pre-defined conversions hide the user-defined conversions in all cases but one:

```
void F(object o, Convertible<object> n) {
    o = n;                          // Pre-defined boxing conversion
    o = (object)n;                  // Pre-defined boxing conversion
    n = o;                          // User-defined implicit conversion
    n = (Convertible<object>)o;     // Pre-defined unboxing conversion
}
```

User-defined conversions are not allowed to convert from or to *interface_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.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

In the example

```
using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

the conversion from `Digit` to `byte` is implicit because it never throws exceptions or loses information, but the conversion from `byte` to `Digit` is explicit since `Digit` can only represent a subset of the possible values of a `byte`.

Instance constructors

An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor_declaration*s:

```
constructor_declaration
    : attributes? constructor_modifier* constructor_declarator constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | constructor_modifier_unsafe
    ;

constructor_declarator
    : identifier '(' formal_parameter_list? ')' constructor_initializer?
    ;

constructor_initializer
    : ':' 'base' '(' argument_list? ')'
    | ':' 'this' '(' argument_list? ')'
    ;

constructor_body
    : block
    | ';'
    ;
```

A *constructor_declaration* may include a set of *attributes* (Attributes), a valid combination of the four access modifiers (Access modifiers), and an `extern` (External methods) 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 (Methods). The formal parameter list defines the signature (Signatures and overloading) of an instance constructor and governs the process whereby overload resolution (Type inference) 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 (Accessibility constraints).

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 Constructor initializers.

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 (Method body).

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 (Default constructors).

Instance constructors are invoked by *object_creation_expressions* (Object creation expressions) and through *constructor_initializers*.

Constructor initializers

All instance constructors (except those for class `object`) implicitly include an invocation of another instance constructor immediately before the *constructor_body*. The constructor to implicitly invoke is determined by the *constructor_initializer*:

- An instance constructor initializer of the form `base(argument_list)` or `base()` causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument_list* if present and the overload resolution rules of Overload resolution. The set of candidate instance constructors consists of all accessible instance constructors contained in the direct base class, or the default constructor (Default constructors), if no instance constructors are 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)` or `this()` causes an instance constructor from the class itself to be invoked. The constructor is selected using *argument_list* if present and the overload resolution rules of Overload resolution. The set of candidate instance constructors consists of all accessible instance constructors declared in the class itself. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

If an instance constructor has no constructor initializer, a constructor initializer of the form `base()` is implicitly provided. Thus, an instance constructor declaration of the form

```
C(...) {...}
```

is exactly equivalent to

```
C(...): base() {...}
```

The scope of the parameters given by the *formal_parameter_list* of an instance constructor declaration includes

the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. For example:

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference `this` in an argument expression of the constructor initializer, as is it a compile-time error for an argument expression to reference any instance member through a *simple_name*.

Instance variable initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable_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.

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.

Given the example

```
using System;

class A
{
    public A() {
        PrintFields();
    }

    public virtual void PrintFields() {}
}

class B: A
{
    int x = 1;
    int y;

    public B() {
        y = -1;
    }

    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

when `new B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 0
```

The value of `x` is 1 because the variable initializer is executed before the base class instance constructor is invoked. However, the value of `y` is 0 (the default value of an `int`) because the assignment to `y` is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor_body*. The example

```
using System;
using System.Collections;

class A
{
    int x = 1, y = -1, count;

    public A() {
        count = 0;
    }

    public A(int n) {
        count = n;
    }
}

class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;

    public B(): this(100) {
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        max = n;
    }
}
```

contains several variable initializers; it also contains constructor initializers of both forms (`base` and `this`). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```
using System.Collections;

class A
{
    int x, y, count;

    public A() {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);                     // Invoke B(int) constructor
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0);     // Variable initializer
        items = new ArrayList(100); // Variable initializer
        A(n - 1);                   // Invoke A(int) constructor
        max = n;
    }
}
```

Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct base class. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where `C` is the name of the class. If overload resolution is unable to determine a unique best candidate for the base class constructor initializer then a compile-time error occurs.

In the example

```
class Message
{
    object sender;
    string text;
}
```

a default constructor is provided because the class contains no instance constructor declarations. Thus, the example is precisely equivalent to

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

Private constructors

When a class `T` declares only private instance constructors, it is not possible for classes outside the program text of `T` to derive from `T` or to directly create instances of `T`. Thus, if a class contains only static members and isn't intended to be instantiated, adding an empty private instance constructor will prevent instantiation. For example:

```
public class Trig
{
    private Trig() {}        // Prevent instantiation

    public const double PI = 3.14159265358979323846;

    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

The `Trig` class groups related methods and constants, but is not intended to be instantiated. Therefore it declares a single empty private instance constructor. At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

Optional instance constructor parameters

The `this(...)` form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters. In the example

```
class Text
{
    public Text(): this(0, 0, null) {}

    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

the first two instance constructors merely provide the default values for the missing arguments. Both use a `this(...)` constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text();                   // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);              // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

Static constructors

A ***static constructor*** is a member that implements the actions required to initialize a closed class type. Static constructors are declared using *static_constructor_declaration*s:

```
static_constructor_declaration
    : attributes? static_constructor_modifiers identifier '(' ')' static_constructor_body
    ;

static_constructor_modifiers
    : 'extern'? 'static'
    | 'static' 'extern'?
    | static_constructor_modifiers_unsafe
    ;

static_constructor_body
    : block
    | ';'
    ;
```

A *static_constructor_declaration* may include a set of *attributes* (Attributes) and an `extern` modifier (External methods).

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 (Method body).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a closed class type executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

- An instance of the class type is created.
- Any of the static members of the class type are referenced.

If a class contains the `Main` method (Application Startup) in which execution begins, the static constructor for that class executes before the `Main` method is called.

To initialize a new closed class type, first a new set of static fields (Static and instance fields) for that particular closed type is created. Each of the static fields is initialized to its default value (Default values). Next, the static field initializers (Static field initialization) are executed for those static fields. Finally, the static constructor is executed.

The example

```
using System;

class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}

class A
{
    static A() {
        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`.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

The example

```
using System;

class A
{
    public static int X;

    static A() {
        X = B.Y + 1;
    }
}

class B
{
    public static int Y = A.X + 1;

    static B() {}

    static void Main() {
        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.

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints (Type parameter constraints). For example, the following type uses a static constructor to enforce that the type argument is an enum:

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a *destructor_declaration*:

```
destructor_declaration
    : attributes? 'extern'? '~' identifier '(' ')' destructor_body
    | destructor_declaration_unsafe
    ;

destructor_body
    : block
    | ';'
    ;
```

A *destructor_declaration* may include a set of *attributes* (Attributes).

The *identifier* of a *destructor_declaration* 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 (Method body).

Destructors are not inherited. Thus, a class has no destructors other than the one which may be declared in that class.

Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in that instance's inheritance chain are called, in order, from most derived to least derived. A destructor may be executed on any thread. For further discussion of the rules that govern when and how a destructor is executed, see Automatic memory management.

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.

Destructors are implemented by overriding the virtual method `Finalize` on `System.Object`. C# programs are not permitted to override this method or call it (or overrides of it) directly. For instance, the program

```
class A
{
    override protected void Finalize() {}    // error

    public void F() {
        this.Finalize();                     // error
    }
}
```

contains two errors.

The compiler behaves as if this method, and overrides of it, do not exist at all. Thus, this program:

```
class A
{
    void Finalize() {}                           // permitted
}
```

is valid, and the method shown hides `System.Object`'s `Finalize` method.

For a discussion of the behavior when an exception is thrown from a destructor, see How exceptions are handled.

Iterators

A function member (Function members) implemented using an iterator block (Blocks) is called an ***iterator***.

An iterator block may be used as the body of a function member as long as the return type of the corresponding function member is one of the enumerator interfaces (Enumerator interfaces) or one of the enumerable interfaces (Enumerable interfaces). It can occur as a *method_body*, *operator_body* or *accessor_body*, whereas events, instance constructors, static constructors and destructors cannot be implemented as iterators.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any `ref` or `out` parameters.

Enumerator interfaces

The ***enumerator interfaces*** are the non-generic interface `System.Collections.IEnumerator` and all instantiations of the generic interface `System.Collections.Generic.IEnumerator<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerator` and `IEnumerator<T>`, respectively.

Enumerable interfaces

The ***enumerable interfaces*** are the non-generic interface `System.Collections.IEnumerable` and all instantiations of the generic interface `System.Collections.Generic.IEnumerable<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerable` and `IEnumerable<T>`, respectively.

Yield type

An iterator produces a sequence of values, all of the same type. This type is called the ***yield type*** of the iterator.

- The yield type of an iterator that returns `IEnumerator` or `IEnumerable` is `object`.
- The yield type of an iterator that returns `IEnumerator<T>` or `IEnumerable<T>` is `T`.

Enumerator objects

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerator object*** is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerator object has the following characteristics:

- It implements `IEnumerator` and `IEnumerator<T>`, where `T` is the yield type of the iterator.
- It implements `System.IDisposable`.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
- It has four potential states, ***before***, ***running***, ***suspended***, and ***after***, and is initially in the ***before*** state.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (Identifiers).

An enumerator object may implement more interfaces than those specified above.

The following sections describe the exact behavior of the `MoveNext`, `Current`, and `Dispose` members of the `IEnumerable` and `IEnumerable<T>` interface implementations provided by an enumerator object.

Note that enumerator objects do not support the `IEnumerator.Reset` method. Invoking this method causes a `System.NotSupportedException` to be thrown.

The MoveNext method

The `MoveNext` method of an enumerator object encapsulates the code of an iterator block. Invoking the `MoveNext` method executes code in the iterator block and sets the `Current` property of the enumerator object as appropriate. The precise action performed by `MoveNext` depends on the state of the enumerator object when `MoveNext` is

invoked:

- If the state of the enumerator object is ***before***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Initializes the parameters (including `this`) of the iterator block to the argument values and instance value saved when the enumerator object was initialized.
  - Executes the iterator block from the beginning until execution is interrupted (as described below).
- If the state of the enumerator object is ***running***, the result of invoking `MoveNext` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Restores the values of all local variables and parameters (including this) to the values saved when execution of the iterator block was last suspended. Note that the contents of any objects referenced by these variables may have changed since the previous call to MoveNext.
  - Resumes execution of the iterator block immediately following the `yield return` statement that caused the suspension of execution and continues until execution is interrupted (as described below).
- If the state of the enumerator object is ***after***, invoking `MoveNext` returns `false`.

When `MoveNext` executes the iterator block, execution can be interrupted in four ways: By a `yield return` statement, by a `yield break` statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

- When a `yield return` statement is encountered (The yield statement):
  - The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
  - Execution of the iterator body is suspended. The values of all local variables and parameters (including `this`) are saved, as is the location of this `yield return` statement. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are not executed at this time.
  - The state of the enumerator object is changed to ***suspended***.
  - The `MoveNext` method returns `true` to its caller, indicating that the iteration successfully advanced to the next value.
- When a `yield break` statement is encountered (The yield statement):
  - If the `yield break` statement is within one or more `try` blocks, the associated `finally` blocks are executed.
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When the end of the iterator body is encountered:
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When an exception is thrown and propagated out of the iterator block:
  - Appropriate `finally` blocks in the iterator body will have been executed by the exception propagation.
  - The state of the enumerator object is changed to ***after***.
  - The exception propagation continues to the caller of the `MoveNext` method.

The Current property

An enumerator object's `Current` property is affected by `yield return` statements in the iterator block.

When an enumerator object is in the ***suspended*** state, the value of `Current` is the value set by the previous call to `MoveNext`. When an enumerator object is in the ***before***, ***running***, or ***after*** states, the result of accessing `Current` is unspecified.

For an iterator with a yield type other than `object`, the result of accessing `Current` through the enumerator

object's `IEnumerable` implementation corresponds to accessing `Current` through the enumerator object's `IEnumerator<T>` implementation and casting the result to `object`.

The Dispose method

The `Dispose` method is used to clean up the iteration by bringing the enumerator object to the ***after*** state.

- If the state of the enumerator object is ***before***, invoking `Dispose` changes the state to ***after***.
- If the state of the enumerator object is ***running***, the result of invoking `Dispose` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `Dispose`:
  - Changes the state to ***running***.
  - Executes any finally blocks as if the last executed `yield return` statement were a `yield break` statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to ***after*** and the exception is propagated to the caller of the `Dispose` method.
  - Changes the state to ***after***.
- If the state of the enumerator object is ***after***, invoking `Dispose` has no affect.

Enumerable objects

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerable object*** is created and returned. The enumerable object's `GetEnumerator` method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerable object has the following characteristics:

- It implements `IEnumerable` and `IEnumerable<T>`, where `T` is the yield type of the iterator.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (Identifiers).

An enumerable object may implement more interfaces than those specified above. In particular, an enumerable object may also implement `IEnumerator` and `IEnumerator<T>`, enabling it to serve as both an enumerable and an enumerator. In that type of implementation, the first time an enumerable object's `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another.

The GetEnumerator method

An enumerable object provides an implementation of the `GetEnumerator` methods of the `IEnumerable` and `IEnumerable<T>` interfaces. The two `GetEnumerator` methods share a common implementation that acquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in Enumerator objects.

Implementation example

This section describes a possible implementation of iterators in terms of standard C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation or the only one possible.

The following `Stack<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the stack in top to bottom order.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```
class Stack<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
        __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
        __state1:
            --i;
            goto __loop;
        __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

In the preceding translation, the code in the iterator block is turned into a state machine and placed in the `MoveNext` method of the enumerator class. Furthermore, the local variable `i` is turned into a field in the enumerator object so it can continue to exist across invocations of `MoveNext`.

The following example prints a simple multiplication table of the integers 1 through 10. The `FromTo` method in the example returns an enumerable object and is implemented using an iterator.

```
using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}
```

The `FromTo` method can be translated into an instantiation of a compiler-generated enumerable class that encapsulates the code in the iterator block, as shown in the following.

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }
```

```
        public int Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
            case 1:
                if (from > to) goto case 2;
                __current = from++;
                __state = 1;
                return true;
            case 2:
                __state = 2;
                return false;
            default:
                throw new InvalidOperationException();
            }
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

The enumerable class implements both the enumerable interfaces and the enumerator interfaces, enabling it to serve as both an enumerable and an enumerator. The first time the `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The `Interlocked.CompareExchange` method is used to ensure thread-safe operation.

The `from` and `to` parameters are turned into fields in the enumerable class. Because `from` is modified in the iterator block, an additional `__from` field is introduced to hold the initial value given to `from` in each enumerator.

The `MoveNext` method throws an `InvalidOperationException` if it is called when `__state` is `0`. This protects against use of the enumerable object as an enumerator object without first calling `GetEnumerator`.

The following example shows a simple tree class. The `Tree<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the tree in infix order.

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main() {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();

        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```
class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> this;
```

```
        Node<T> __this;
        IEnumerator<T> __left, __right;
        int __state;
        T __current;

        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            try {
                switch (__state) {

                case 0:
                    __state = -1;
                    if (__this.left == null) goto __yield_value;
                    __left = __this.left.GetEnumerator();
                    goto case 1;

                case 1:
                    __state = -2;
                    if (!__left.MoveNext()) goto __left_dispose;
                    __current = __left.Current;
                    __state = 1;
                    return true;

                __left_dispose:
                    __state = -1;
                    __left.Dispose();

                __yield_value:
                    __current = __this.value;
                    __state = 2;
                    return true;

                case 2:
                    __state = -1;
                    if (__this.right == null) goto __end;
                    __right = __this.right.GetEnumerator();
                    goto case 3;

                case 3:
                    __state = -3;
                    if (!__right.MoveNext()) goto __right_dispose;
                    __current = __right.Current;
                    __state = 3;
                    return true;

                __right_dispose:
                    __state = -1;
                    __right.Dispose();

                __end:
                    __state = 4;
                    break;

                }
            }
            finally {
                if (__state < 0) Dispose();
            }
            return false;
```

```
            return false;
        }

        public void Dispose() {
            try {
                switch (__state) {

                case 1:
                case -2:
                    __left.Dispose();
                    break;

                case 3:
                case -3:
                    __right.Dispose();
                    break;

                }
            }
            finally {
                __state = 4;
            }
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

The compiler generated temporaries used in the `foreach` statements are lifted into the `__left` and `__right` fields of the enumerator object. The `__state` field of the enumerator object is carefully updated so that the correct `Dispose()` method will be called correctly if an exception is thrown. Note that it is not possible to write the translated code with simple `foreach` statements.

Async functions

A method (Methods) or anonymous function (Anonymous function expressions) with the `async` modifier is called an ***async function***. In general, the term ***async*** is used to describe any kind of function that has the `async` modifier.

It is a compile-time error for the formal parameter list of an async function to specify any `ref` or `out` parameters.

The *return_type* of an async method must be either `void` or a ***task type***. The task types are `System.Threading.Tasks.Task` and types constructed from `System.Threading.Tasks.Task<T>`. For the sake of brevity, in this chapter these types are referenced as `Task` and `Task<T>`, respectively. An async method returning a task type is said to be task-returning.

The exact definition of the task types is implementation defined, but from the language's point of view a task type is in one of the states incomplete, succeeded or faulted. A faulted task records a pertinent exception. A succeeded `Task<T>` records a result of type `T`. Task types are awaitable, and can therefore be the operands of await expressions (Await expressions).

An async function invocation has the ability to suspend evaluation by means of await expressions (Await expressions) in its body. Evaluation may later be resumed at the point of the suspending await expression by means of a ***resumption delegate***. The resumption delegate is of type `System.Action`, and when it is invoked, evaluation of the async function invocation will resume from the await expression where it left off. The ***current caller*** of an async function invocation is the original caller if the function invocation has never been suspended, or the most recent caller of the resumption delegate otherwise.

Evaluation of a task-returning async function

Invocation of a task-returning async function causes an instance of the returned task type to be generated. This is

called the ***return task*** of the async function. The task is initially in an incomplete state.

The async function body is then evaluated until it is either suspended (by reaching an await expression) or terminates, at which point control is returned to the caller, along with the return task.

When the body of the async function terminates, the return task is moved out of the incomplete state:

- If the function body terminates as the result of reaching a return statement or the end of the body, any result value is recorded in the return task, which is put into a succeeded state.
- If the function body terminates as the result of an uncaught exception (The throw statement) the exception is recorded in the return task which is put into a faulted state.

Evaluation of a void-returning async function

If the return type of the async function is `void`, evaluation differs from the above in the following way: Because no task is returned, the function instead communicates completion and exceptions to the current thread's ***synchronization context***. The exact definition of synchronization context is implementation-dependent, but is a representation of "where" the current thread is running. The synchronization context is notified when evaluation of a void-returning async function commences, completes successfully, or causes an uncaught exception to be thrown.

This allows the context to keep track of how many void-returning async functions are running under it, and to decide how to propagate exceptions coming out of them.

Structs

1/13/2018 • 19 minutes to read • Edit Online

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

As described in Simple types, 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 (Struct examples).

Struct declarations

A *struct_declaration* is a *type_declaration* (Type declarations) that declares a new struct:

```
struct_declaration
    : attributes? struct_modifier* 'partial'? 'struct' identifier type_parameter_list?
      struct_interfaces? type_parameter_constraints_clause* struct_body ';'?
    ;
```

A *struct_declaration* consists of an optional set of *attributes* (Attributes), followed by an optional set of *struct_modifiers* (Struct modifiers), followed by an optional `partial` modifier, followed by the keyword `struct` and an *identifier* that names the struct, followed by an optional *type_parameter_list* specification (Type parameters), followed by an optional *struct_interfaces* specification (Partial modifier)), followed by an optional *type_parameter_constraints_clauses* specification (Type parameter constraints), followed by a *struct_body* (Struct body), optionally followed by a semicolon.

Struct modifiers

A *struct_declaration* may optionally include a sequence of struct modifiers:

```
struct_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | struct_modifier_unsafe
    ;
```

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 (Class declarations).

Partial modifier

The `partial` modifier indicates that this *struct_declaration* is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in Partial types.

Struct interfaces

A struct declaration may include a *struct_interfaces* specification, in which case the struct is said to directly implement the given interface types.

```
struct_interfaces
    : ':' interface_type_list
    ;
```

Interface implementations are discussed further in Interface implementations.

Struct body

The *struct_body* of a struct defines the members of the struct.

```
struct_body
    : '{' struct_member_declaration* '}'
    ;
```

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_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | static_constructor_declaration
    | type_declaration
    | struct_member_declaration_unsafe
    ;
```

Except for the differences noted in Class and struct differences, the descriptions of class members provided in Class members through Iterators apply to struct members as well.

Class and struct differences

Structs differ from classes in several important ways:

- Structs are value types (Value semantics).
- All struct types implicitly inherit from the class `System.ValueType` (Inheritance).
- Assignment to a variable of a struct type creates a copy of the value being assigned (Assignment).
- 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` (Default values).
- Boxing and unboxing operations are used to convert between a struct type and `object` (Boxing and unboxing).
- The meaning of `this` is different for structs (This access).

- Instance field declarations for a struct are not permitted to include variable initializers (Field initializers).
- A struct is not permitted to declare a parameterless instance constructor (Constructors).
- A struct is not permitted to declare a destructor (Destructors).

Value semantics

Structs are value types (Value types) and are said to have value semantics. Classes, on the other hand, are reference types (Reference types) and are said to have reference semantics.

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object. When a struct `B` contains an instance field of type `A` and `A` is a struct type, it is a compile-time error for `A` to depend on `B` or a type constructed from `B`. A struct `X` ***directly depends on*** a struct `Y` if `X` contains an instance field of type `Y`. Given this definition, the complete set of structs upon which a struct depends is the transitive closure of the ***directly depends on*** relationship. For example

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

is an error because `Node` contains an instance field of its own type. Another example

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

is an error because each of the types `A`, `B`, and `C` depend on each other.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data (except in the case of `ref` and `out` parameter variables), and it is not possible for operations on one to affect the other. Furthermore, because structs are not reference types, it is not possible for values of a struct type to be `null`.

Given the declaration

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

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

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

outputs the value `10`. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the

assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

Inheritance

All struct types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. A struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration to specify a base class.

Struct types are never abstract and are always implicitly sealed. The `abstract` and `sealed` modifiers are therefore not permitted in a struct declaration.

Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be `protected` or `protected internal`.

Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed only to override methods inherited from `System.ValueType`.

Assignment

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

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in Simple assignment.

Default values

As described in Default values, several kinds of variables are automatically initialized to their default value when they are created. For variables of class types and other reference types, this default value is `null`. However, since structs are value types that cannot be `null`, the default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null`.

Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero.

The default value of a struct corresponds to the value returned by the default constructor of the struct (Default 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`.

Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state.

Boxing and unboxing

A value of a class type can be converted to type `object` or to an interface type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a value of type `object` or a value of an interface type can be converted back to a class type without changing the reference (but of course a run-time type check is required in this case).

Since structs are not reference types, these operations are implemented differently for struct types. When a value of a struct type is converted to type `object` or to an interface type that is implemented by the struct, a boxing operation takes place. Likewise, when a value of type `object` or a value of an interface type is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing copies the struct value either into or out of the boxed instance. Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct.

When a struct type overrides a virtual method inherited from `System.Object` (such as `Equals`, `GetHashCode`, or `ToString`), invocation of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example:

```
using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main() {
        Test<Counter>();
    }
}
```

The output of the program is:

```
1
2
3
```

Although it is bad style for `ToString` to have side effects, the example demonstrates that no boxing occurred for the three invocations of `x.ToString()`.

Similarly, boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a method `Increment` which can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on, never a boxed copy.

```
using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();                    // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment();        // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}
```

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus, the output of the program is:

```
0
1
1
```

For further details on boxing and unboxing, see Boxing and unboxing.

Meaning of this

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class.

Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type, and within an instance function member of a struct, `this` corresponds to a `ref` parameter of the struct type. In both cases, `this` is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to `this` or by passing this as a `ref` or `out` parameter.

Field initializers

As described in Default values, the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to `null`. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

The example

```
struct Point
{
    public int x = 1;  // Error, initializer not permitted
    public int y = 1;  // Error, initializer not permitted
}
```

is in error because the instance field declarations include variable initializers.

Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null (Default constructors). A struct can declare instance constructors having parameters. For example

```
struct Point
{
    int x, y;

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

Given the above declaration, the statements

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

both create a `Point` with `x` and `y` initialized to zero.

A struct instance constructor is not permitted to include a constructor initializer of the form `base(...)`.

If the struct instance constructor doesn't specify a constructor initializer, the `this` variable corresponds to an `out` parameter of the struct type, and similar to an `out` parameter, `this` must be definitely assigned (Definite assignment) at every location where the constructor returns. If the struct instance constructor specifies a constructor initializer, the `this` variable corresponds to a `ref` parameter of the struct type, and similar to a `ref` parameter, `this` is considered definitely assigned on entry to the constructor body. Consider the instance constructor implementation below:

```
struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;        // error, this is not yet definitely assigned
        Y = y;        // error, this is not yet definitely assigned
    }
}
```

No instance member function (including the set accessors for the properties `X` and `Y`) can be called until all fields of the struct being constructed have been definitely assigned. The only exception involves automatically implemented properties (Automatically implemented properties). The definite assignment rules (Simple assignment expressions) specifically exempt assignment to an auto-property of a struct type within an instance constructor of that struct type: such an assignment is considered a definite assignment of the hidden backing field of the auto-property. Thus, the following is allowed:

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
        X = x;      // allowed, definitely assigns backing field
        Y = y;      // allowed, definitely assigns backing field
    }
```

Destructors

A struct is not permitted to declare a destructor.

Static constructors

Static constructors for structs follow most of the same rules as for classes. The execution of a static constructor for a struct type is triggered by the first of the following events to occur within an application domain:

- A static member of the struct type is referenced.
- An explicitly declared constructor of the struct type is called.

The creation of default values (Default values) of struct types does not trigger the static constructor. (An example of this is the initial value of elements in an array.)

Struct examples

The following shows two significant examples of using `struct` types to create types that can be used similarly to the predefined types of the language, but with modified semantics.

Database integer type

The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```
using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();

    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.

    int value;
    bool defined;

    // Private instance constructor. Creates a DBInt with a known value.

    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }

    // The IsNull property is true if this DBInt represents an unknown value.

    public bool IsNull { get { return !defined; } }

    // The Value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.

    public int Value { get { return value; } }

    // Implicit conversion from int to DBInt.

    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }

    // Explicit conversion from DBInt to int. Throws an exception if the
    // given DBInt represents an unknown value.

    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }

    public static DBInt operator +(DBInt x) {
        return x;
    }

    public static DBInt operator -(DBInt x) {
        return x.defined ? -x.value : Null;
    }

    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value + y.value: Null;
    }

    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value - y.value: Null;
    }

    public static DBInt operator *(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value * y.value: Null;
    }

    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value / y.value: Null;
    }
```

```
    public static DBInt operator %(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value % y.value: Null;
    }

    public static DBBool operator ==(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value == y.value: DBBool.Null;
    }

    public static DBBool operator !=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value != y.value: DBBool.Null;
    }

    public static DBBool operator >(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value > y.value: DBBool.Null;
    }

    public static DBBool operator <(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }

    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }

    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        return defined? value.ToString(): "DBInt.Null";
    }
}
```

Database boolean type

The `DBBool` struct below implements a three-valued logical type. The possible values of this type are `DBBool.True`, `DBBool.False`, and `DBBool.Null`, where the `Null` member indicates an unknown value. Such three-valued logical types are commonly used in databases.

```
using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or 1.

    DBBool(int value) {
```

```
        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.

    public bool IsNull { get { return value == 0; } }

    public bool IsFalse { get { return value < 0; } }

    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.

    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null, otherwise returns true or false.

    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.

    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }

    // Inequality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.

    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }

    // Logical negation operator. Returns True if the operand is False, Null
    // if the operand is Null, or False if the operand is True.

    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }

    // Logical AND operator. Returns False if either operand is False,
    // otherwise Null if either operand is Null, otherwise True.

    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }

    // Logical OR operator. Returns True if either operand is True, otherwise
    // Null if either operand is Null, otherwise False.

    public static DBBool operator |(DBBool x, DBBool y) {
        return new DBBool(x.value > y.value? x.value: y.value);
    }

    // Definitely true operator. Returns true if the operand is True, false
    // otherwise.

    public static bool operator true(DBBool x) {
        return x.value > 0;
```

```
        return x.value > 0;
    }

    // Definitely false operator. Returns true if the operand is False, false
    // otherwise.

    public static bool operator false(DBBool x) {
        return x.value < 0;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

Arrays

1/13/2018 • 9 minutes to read • Edit Online

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

An array has a rank which determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of one is called a ***single-dimensional array***. An array with a rank greater than one is called a ***multi-dimensional array***. Specific sized multi-dimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so on.

Each dimension of an array has an associated length which is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length `N`, indices can range from `0` to `N - 1` inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

The element type of an array can be any type, including an array type.

Array types

An array type is written as a *non_array_type* followed by one or more *rank_specifier*s:

```
array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;
```

A *non_array_type* is any *type* that is not itself an *array_type*.

The rank of an array type is given by the leftmost *rank_specifier* in the *array_type*: A *rank_specifier* indicates that the array is an array with a rank of one plus the number of "`,`" tokens in the *rank_specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank_specifier*:

- An array type of the form `T[R]` is an array with rank `R` and a non-array element type `T`.
- An array type of the form `T[R][R1]...[Rn]` is an array with rank `R` and an element type `T[R1]...[Rn]`.

In effect, the *rank_specifier*s are read from left to right before the final non-array element type. The type `int[][,,][,]` is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of `int`.

At run-time, a value of an array type can be `null` or a reference to an instance of that array type.

The System.Array type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion (Implicit reference conversions) exists from any array type to `System.Array`, and an explicit reference conversion (Explicit reference conversions) 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.

Arrays and the generic IList interface

A one-dimensional array `T[]` implements the interface `System.Collections.Generic.IList<T>` (`IList<T>` for short) and its base interfaces. Accordingly, there is an implicit conversion from `T[]` to `IList<T>` and its base interfaces. In addition, if there is an implicit reference conversion from `S` to `T` then `S[]` implements `IList<T>` and there is an implicit reference conversion from `S[]` to `IList<T>` and its base interfaces (Implicit reference conversions). If there is an explicit reference conversion from `S` to `T` then there is an explicit reference conversion from `S[]` to `IList<T>` and its base interfaces (Explicit reference conversions). For example:

```
using System.Collections.Generic;

class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;                    // Ok
        IList<string> lst2 = oa1;                   // Error, cast needed
        IList<object> lst3 = sa;                    // Ok
        IList<object> lst4 = oa1;                   // Ok

        IList<string> lst5 = (IList<string>)oa1;    // Exception
        IList<string> lst6 = (IList<string>)oa2;    // Ok
    }
}
```

The assignment `lst2 = oa1` generates a compile-time error since the conversion from `object[]` to `IList<string>` is an explicit conversion, not implicit. The cast `(IList<string>)oa1` will cause an exception to be thrown at run-time since `oa1` references an `object[]` and not a `string[]`. However the cast `(IList<string>)oa2` will not cause an exception to be thrown since `oa2` references a `string[]`.

Whenever there is an implicit or explicit reference conversion from `S[]` to `IList<T>`, there is also an explicit reference conversion from `IList<T>` and its base interfaces to `S[]` (Explicit reference conversions).

When an array type `S[]` implements `IList<T>`, some of the members of the implemented interface may throw exceptions. The precise behavior of the implementation of the interface is beyond the scope of this specification.

Array creation

Array instances are created by *array_creation_expression*s (Array creation expressions) or by field or local variable declarations that include an *array_initializer* (Array initializers).

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 (Default

values).

Array element access

Array elements are accessed using *element_access* expressions (Array access) of the form `A[I1, I2, ..., In]`, where `A` is an expression of an array type and each `Ix` is an expression of type `int`, `uint`, `long`, `ulong`, or can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement (The foreach statement).

Array members

Every array type inherits the members declared by the `System.Array` type.

Array covariance

For any two *reference_types* `A` and `B`, if an implicit reference conversion (Implicit reference conversions) or explicit reference conversion (Explicit reference conversions) 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 (Simple assignment). For example:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

The assignment to `array[i]` in the `Fill` method implicitly includes a run-time check which ensures that the object referenced by `value` is either `null` or an instance that is compatible with the actual element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array.

Array covariance specifically does not extend to arrays of *value_types*. For example, no conversion exists that permits an `int[]` to be treated as an `object[]`.

Array initializers

Array initializers may be specified in field declarations (Fields), local variable declarations (Local variable declarations), and array creation expressions (Array creation expressions):

```
array_initializer
    : '{' variable_initializer_list? '}'
    | '{' variable_initializer_list ',' '}'
    ;

variable_initializer_list
    : variable_initializer (',' variable_initializer)*
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

An array initializer consists of a sequence of variable initializers, enclosed by "`{`" and "`}`" tokens and separated by "`,`" tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer, or is inferred from the expressions in the array initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

If a dimension other than the rightmost is given with length zero, the subsequent dimensions are assumed to also have length zero. The example:

```
int[,] c = {};
```

creates a two-dimensional array with a length of zero for both the leftmost and the rightmost dimension:

```
int[,] c = new int[0, 0];
```

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Here are some examples:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};        // OK
int[] y = new int[i] {0, 1, 2};        // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};     // Error, length/initializer mismatch
```

Here, the initializer for `y` results in a compile-time error because the dimension length expression is not a constant, and the initializer for `z` results in a compile-time error because the length and the number of elements in the initializer do not agree.

Interfaces

1/26/2018 • 29 minutes to read • Edit Online

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

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

Interface declarations

An *interface_declaration* is a *type_declaration* (Type declarations) that declares a new interface type.

```
interface_declaration
    : attributes? interface_modifier* 'partial'? 'interface'
      identifier variant_type_parameter_list? interface_base?
      type_parameter_constraints_clause* interface_body ';'?
    ;
```

An *interface_declaration* consists of an optional set of *attributes* (Attributes), followed by an optional set of *interface_modifiers* (Interface modifiers), followed by an optional `partial` modifier, followed by the keyword `interface` and an *identifier* that names the interface, followed by an optional *variant_type_parameter_list* specification (Variant type parameter lists), followed by an optional *interface_base* specification (Base interfaces), followed by an optional *type_parameter_constraints_clauses* specification (Type parameter constraints), followed by an *interface_body* (Interface body), optionally followed by a semicolon.

Interface modifiers

An *interface_declaration* may optionally include a sequence of interface modifiers:

```
interface_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | interface_modifier_unsafe
    ;
```

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The `new` modifier is only permitted on interfaces defined within a class. It specifies that the interface hides an inherited member by the same name, as described in The new modifier.

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 (Declared accessibility).

Partial modifier

The `partial` modifier indicates that this *interface_declaration* is a partial type declaration. Multiple partial interface declarations with the same name within an enclosing namespace or type declaration combine to form one interface declaration, following the rules specified in Partial types.

Variant type parameter lists

Variant type parameter lists can only occur on interface and delegate types. The difference from ordinary *type_parameter_list*s is the optional *variance_annotation* on each type parameter.

```
variant_type_parameter_list
    : '<' variant_type_parameters '>'
    ;

variant_type_parameters
    : attributes? variance_annotation? type_parameter
    | variant_type_parameters ',' attributes? variance_annotation? type_parameter
    ;

variance_annotation
    : 'in'
    | 'out'
    ;
```

If the variance annotation is `out`, the type parameter is said to be ***covariant***. If the variance annotation is `in`, the type parameter is said to be ***contravariant***. If there is no variance annotation, the type parameter is said to be ***invariant***.

In the example

```
interface C<out X, in Y, Z>
{
  X M(Y y);
  Z P { get; set; }
}
```

`X` is covariant, `Y` is contravariant and `Z` is invariant.

Variance safety

The occurrence of variance annotations in the type parameter list of a type restricts the places where types can occur within the type declaration.

A type `T` is ***output-unsafe*** if one of the following holds:

- `T` is a contravariant type parameter
- `T` is an array type with an output-unsafe element type
- `T` is an interface or delegate type `S<A1,...,Ak>` constructed from a generic type `S<X1,...,Xk>` where for at least one `Ai` one of the following holds:
  - `Xi` is covariant or invariant and `Ai` is output-unsafe.
  - `Xi` is contravariant or invariant and `Ai` is input-safe.

A type `T` is ***input-unsafe*** if one of the following holds:

- `T` is a covariant type parameter
- `T` is an array type with an input-unsafe element type
- `T` is an interface or delegate type `S<A1,...,Ak>` constructed from a generic type `S<X1,...,Xk>` where for at least one `Ai` one of the following holds:
  - `Xi` is covariant or invariant and `Ai` is input-unsafe.
  - `Xi` is contravariant or invariant and `Ai` is output-unsafe.

Intuitively, an output-unsafe type is prohibited in an output position, and an input-unsafe type is prohibited in an input position.

A type is ***output-safe*** if it is not output-unsafe, and ***input-safe*** if it is not input-unsafe.

Variance conversion

The purpose of variance annotations is to provide for more lenient (but still type safe) conversions to interface and delegate types. To this end the definitions of implicit (Implicit conversions) and explicit conversions (Explicit conversions) make use of the notion of variance-convertibility, which is defined as follows:

A type `T<A1,...,An>` is variance-convertible to a type `T<B1,...,Bn>` if `T` is either an interface or a delegate type declared with the variant type parameters `T<X1,...,Xn>`, and for each variant type parameter `Xi` one of the following holds:

- `Xi` is covariant and an implicit reference or identity conversion exists from `Ai` to `Bi`
- `Xi` is contravariant and an implicit reference or identity conversion exists from `Bi` to `Ai`
- `Xi` is invariant and an identity conversion exists from `Ai` to `Bi`

Base interfaces

An interface can inherit from zero or more interface types, which are called the ***explicit base interfaces*** of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma separated list of base interface types.

```
interface_base
    : ':' interface_type_list
    ;
```

For a constructed interface type, the explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each *type_parameter* in the base interface declaration, the corresponding *type_argument* of the constructed type.

The explicit base interfaces of an interface must be at least as accessible as the interface itself (Accessibility constraints). For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface_base* of a `public` interface.

It is a compile-time error for an interface to directly or indirectly inherit from itself.

The ***base interfaces*** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

Every base interface of an interface must be output-safe (Variance safety). A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

Interface body

The *interface_body* of an interface defines the members of the interface.

```
interface_body
    : '{' interface_member_declaration* '}'
    ;
```

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_declaration
    : interface_method_declaration
    | interface_property_declaration
    | interface_event_declaration
    | interface_indexer_declaration
    ;
```

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

The example

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An *interface_declaration* creates a new declaration space (Declarations), and the *interface_member_declaration*s 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 (Signatures and overloading) of a method must differ from the signatures of all other methods declared in the same interface, and two methods declared in the same interface may not have signatures that differ solely by `ref` and `out`.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in Hiding through inheritance.

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

Note that the members in class `object` are not, strictly speaking, members of any interface (Interface members). However, the members in class `object` are available via member lookup in any interface type (Member lookup).

Interface methods

Interface methods are declared using *interface_method_declarations*:

```
interface_method_declaration
    : attributes? 'new'? return_type identifier type_parameter_list
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;
```

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 (Methods). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

Each formal parameter type of an interface method must be input-safe (Variance safety), and the return type must be either `void` or output-safe. Furthermore, each class type constraint, interface type constraint and type parameter constraint on any type parameter of the method must be input-safe.

These rules ensure that any covariant or contravariant usage of the interface remains type-safe. For example,

```
interface I<out T> { void M<U>() where U : T; }
```

is illegal because the usage of `T` as a type parameter constraint on `U` is not input-safe.

Were this restriction not in place it would be possible to violate type safety in the following manner:

```
class B {}
class D : B{}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

This is actually a call to `C.M<E>`. But that call requires that `E` derive from `D`, so type safety would be violated here.

Interface properties

Interface properties are declared using *interface_property_declarations*:

```
interface_property_declaration
    : attributes? 'new'? type identifier '{' interface_accessors '}'
    ;

interface_accessors
    : attributes? 'get' ';'
    | attributes? 'set' ';'
    | attributes? 'get' ';' attributes? 'set' ';'
    | attributes? 'set' ';' attributes? 'get' ';'
    ;
```

The *attributes*, *type*, and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class (Properties).

The accessors of an interface property declaration correspond to the accessors of a class property declaration (Accessors), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

The type of an interface property must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

Interface events

Interface events are declared using *interface_event_declaration*s:

```
interface_event_declaration
    : attributes? 'new'? '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 (Events).

The type of an interface event must be input-safe.

Interface indexers

Interface indexers are declared using *interface_indexer_declaration*s:

```
interface_indexer_declaration
    : attributes? 'new'? 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 (Indexers).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (Indexers), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

All the formal parameter types of an interface indexer must be input-safe . In addition, any `out` or `ref` formal parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

The type of an interface indexer must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

Interface member access

Interface members are accessed through member access (Member access) and indexer access (Indexer access) expressions of the form `I.M` and `I[A]`, where `I` is an interface type, `M` is a method, property, or event of that

interface type, and `A` is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup (Member lookup), method invocation (Method invocations), and indexer access (Indexer access) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be used to resolve the ambiguities.

In the example

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}

class C
{
    void Test(IListCounter x) {
        x.Count(1);                 // Error
        x.Count = 1;                // Error
        ((IList)x).Count = 1;       // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1);     // Ok, invokes ICounter.Count
    }
}
```

the first two statements cause compile-time errors because the member lookup (Member lookup) of `Count` in `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time.

In the example

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);               // Invokes IInteger.Add
        n.Add(1.0);             // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);   // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);    // Only IDouble.Add is a candidate
    }
}
```

the invocation `n.Add(1)` selects `IInteger.Add` by applying the overload resolution rules of Overload resolution. Similarly the invocation `n.Add(1.0)` selects `IDouble.Add`. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.

In the example

```
interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                 // Invokes ILeft.F
        ((IBase)d).F(1);        // Invokes IBase.F
        ((ILeft)d).F(1);        // Invokes ILeft.F
        ((IRight)d).F(1);       // Invokes IBase.F
    }
}
```

the `IBase.F` member is hidden by the `ILeft.F` member. The invocation `d.F(1)` thus selects `ILeft.F`, even though `IBase.F` appears to not be hidden in the access path that leads through `IRight`.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from `IDerived` to `ILeft` to `IBase` hides `IBase.F`, the member is also hidden in the access path from `IDerived` to `IRight` to `IBase`.

Fully qualified interface member names

An interface member is sometimes referred to by its ***fully qualified name***. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. For example, given the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}
```

the fully qualified name of `Paint` is `IControl.Paint` and the fully qualified name of `SetText` is `ITextBox.SetText`.

In the example above, it is not possible to refer to `Paint` as `ITextBox.Paint`.

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. For example

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`.

Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

A class or struct that directly implements an interface also directly implements all of the interface's base interfaces implicitly. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. For example:

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Here, class `TextBox` implements both `IControl` and `ITextBox`.

When a class `C` directly implements an interface, all classes derived from C also implement the interface implicitly. The base interfaces specified in a class declaration can be constructed interface types (Constructed types). A base interface cannot be a type parameter on its own, though it can involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types:

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in Uniqueness of implemented interfaces.

Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare ***explicit interface member implementations***. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

Here `IDictionary<int,T>.this` and `IDictionary<int,T>.Add` are explicit interface member implementations.

In some cases, the name of an interface member may not be appropriate for the implementing class, in which case

the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a `Close` member function that has the effect of releasing the file resource, and implement the `Dispose` method of the `IDisposable` interface using explicit interface member implementation:

```
interface IDisposable
{
    void Dispose();
}

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can only be accessed through an interface instance, and is in that case referenced simply by its member name.

It is a compile-time error for an explicit interface member implementation to include access modifiers, and it is a compile-time error to include the modifiers `abstract`, `virtual`, `override`, or `static`.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through their fully qualified name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense also public.

Explicit interface member implementations serve two primary purposes:

- Because explicit interface member implementations are not accessible through class or struct instances, they allow interface implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of that class or struct.
- Explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the following class

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}    // invalid
}
```

the declaration of `IComparable.CompareTo` results in a compile-time error because `IComparable` is not listed in the

base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the declarations

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}

class Ellipse: Shape
{
    object ICloneable.Clone() {...}    // invalid
}
```

the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is not explicitly listed in the base class list of `Ellipse`.

The fully qualified name of an interface member must reference the interface in which the member was declared. Thus, in the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

the explicit interface member implementation of `Paint` must be written as `IControl.Paint`.

Uniqueness of implemented interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows:

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>                    // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Were this permitted, it would be impossible to determine which code to execute in the following case:

```
I<int> x = new X<int,int>();
x.F();
```

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

- Let `L` be the list of interfaces directly specified in a generic class, struct, or interface declaration `C`.
- Add to `L` any base interfaces of the interfaces already in `L`.
- Remove any duplicates from `L`.
- If any possible constructed type created from `C` would, after type arguments are substituted into `L`, cause two interfaces in `L` to be identical, then the declaration of `C` is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration `X` above, the interface list `L` consists of `I<U>` and `I<V>`. The declaration is invalid because any constructed type with `U` and `V` being the same type would cause these two interfaces to be identical types.

It is possible for interfaces specified at different inheritance levels to unify:

```
interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>    // Ok
{
    void I<V>.F() {...}
}
```

This code is valid even though `Derived<U,V>` implements both `I<U>` and `I<V>`. The code

```
I<int> x = new Derived<int,int>();
x.F();
```

invokes the method in `Derived`, since `Derived<int,int>` effectively re-implements `I<int>` (Interface re-implementation).

Implementation of generic methods

When a generic method implicitly implements an interface method, the constraints given for each method type parameter must be equivalent in both declarations (after any interface type parameters are replaced with the appropriate type arguments), where method type parameters are identified by ordinal positions, left to right.

When a generic method explicitly implements an interface method, however, no constraints are allowed on the implementing method. Instead, the constraints are inherited from the interface method

```
interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
    public void F<T>(T t) {...}                    // Ok
    public void G<T>(T t) where T: C {...}         // Ok
    public void H<T>(T t) where T: string {...}    // Error
}
```

The method `C.F<T>` implicitly implements `I<object,C,string>.F<T>`. In this case, `C.F<T>` is not required (nor permitted) to specify the constraint `T:object` since `object` is an implicit constraint on all type parameters. The method `C.G<T>` implicitly implements `I<object,C,string>.G<T>` because the constraints match those in the interface, after the interface type parameters are replaced with the corresponding type arguments. The constraint for method `C.H<T>` is an error because sealed types (`string` in this case) cannot be used as constraints. Omitting the constraint would also be an error since constraints of implicit interface method implementations are required to match. Thus, it is impossible to implicitly implement `I<object,C,string>.H<T>`. This interface method can only be implemented using an explicit interface member implementation:

```
class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;    // Ok
        H<T>(t);
    }
}
```

In this example, the explicit interface member implementation invokes a public method having strictly weaker constraints. Note that the assignment from `t` to `s` is valid since `T` inherits a constraint of `T:string`, even though this constraint is not expressible in source code.

Interface mapping

A class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as ***interface mapping***.

Interface mapping for a class or struct `C` locates an implementation for each member of each interface specified in the base class list of `C`. The implementation of a particular interface member `I.M`, where `I` is the interface in which the member `M` is declared, is determined by examining each class or struct `S`, starting with `C` and repeating for each successive base class of `C`, until a match is located:

- If `S` contains a declaration of an explicit interface member implementation that matches `I` and `M`, then this member is the implementation of `I.M`.
- Otherwise, if `S` contains a declaration of a non-static public member that matches `M`, then this member is the implementation of `I.M`. If more than one member matches, it is unspecified which member is the implementation of `I.M`. This situation can only occur if `S` is a constructed type where the two members as declared in the generic type have different signatures, but the type arguments make their signatures identical.

A compile-time error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of `C`. Note that the members of an interface include those members that are inherited from base interfaces.

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
- `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Neither non-public nor static members participate in interface mapping.

In the example

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

the `ICloneable.Clone` member of `C` becomes the implementation of `Clone` in `ICloneable` because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. For example

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

Here, the `Paint` methods of both `IControl` and `IForm` are mapped onto the `Paint` method in `Page`. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For example

```
interface IBase
{
    int P { get; }
}

interface IDerived: IBase
{
    new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}
```

it is not possible to have separate implementations for the `IControl` named in the base class list, the `IControl` inherited by `ITextBox`, and the `IControl` inherited by `IListBox`. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of `ITextBox` and `IListBox` share the same implementation of `IControl`, and `ComboBox` is simply considered to implement three interfaces, `IControl`, `ITextBox`, and `IListBox`.

The members of a base class participate in interface mapping. In the example

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

class Class2: Class1, Interface1
{
    new public void G() {}
}
```

the method `F` in `Class1` is used in `Class2`'s implementation of `Interface1`.

Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly ***re-implementing*** an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public void Paint() {...}
}

class TextBox: Control
{
    new public void Paint() {...}
}
```

the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface instances will have the following effects

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();            // invokes Control.Paint();
t.Paint();            // invokes TextBox.Paint();
ic.Paint();           // invokes Control.Paint();
it.Paint();           // invokes Control.Paint();
```

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the declarations above to

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

the following effects will now be observed

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();            // invokes Control.Paint();
t.Paint();            // invokes TextBox.Paint();
ic.Paint();           // invokes Control.Paint();
it.Paint();           // invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. For example

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Here, classes derived from `Control` can specialize the implementation of `IControl.Paint` by overriding the `PaintControl` method.

Interface re-implementation

A class that inherits an interface implementation is permitted to ***re-implement*** the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

the fact that `Control` maps `IControl.Paint` onto `Control.IControl.Paint` doesn't affect the re-implementation in `MyControl`, which maps `IControl.Paint` onto `MyControl.Paint`.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. For example

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}
```

Here, the implementation of `IMethods` in `Derived` maps the interface methods onto `Derived.F`, `Base.IMethods.G`, `Derived.IMethods.H`, and `Base.I`.

When a class implements an interface, it implicitly also implements all of that interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. For example

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}
```

Here, the re-implementation of `IDerived` also re-implements `IBase`, mapping `IBase.F` onto `D.F`.

Abstract classes and interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. For example

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Here, the implementation of `IMethods` maps `F` and `G` onto abstract methods, which must be overridden in non-abstract classes that derive from `C`.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. For example

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

Here, non-abstract classes that derive from `C` would be required to override `FF` and `GG`, thus providing the actual implementation of `IMethods`.

Enums

1/13/2018 • 6 minutes to read • Edit Online

An ***enum type*** is a distinct value type (Value types) that declares a set of named constants.

The example

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

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

Enum declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and defines the name, accessibility, underlying type, and members of the enum.

```
enum_declaration
    : attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
    ;

enum_base
    : ':' integral_type
    ;

enum_body
    : '{' enum_member_declarations? '}'
    | '{' enum_member_declarations ',' '}'
    ;
```

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Note that `char` cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

The example

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

declares an enum with an underlying type of `long`. A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future.

Enum modifiers

An *enum_declaration* may optionally include a sequence of enum modifiers:

```
enum_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 (Class modifiers). Note, however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

Enum members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

```
enum_member_declarations
    : enum_member_declaration (',' enum_member_declaration)*
    ;

enum_member_declaration
    : attributes? identifier ('=' constant_expression)?
    ;
```

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the underlying integral type `uint`.

Multiple enum members may share the same associated value. The example

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

    Max = Blue
}
```

shows an enum in which two enum members -- `Blue` and `Max` -- have the same associated value.

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant_expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is zero.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by one. This increased value must be within the range of values that can be represented by the underlying type, otherwise a compile-time error occurs.

The example

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}

class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);

            case Color.Green:
                return String.Format("Green = {0}", (int) c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);

            default:
                return "Invalid color";
        }
    }
}
```

prints out the enum member names and their associated values. The output is:

```
Red = 0
Green = 10
Blue = 11
```

for the following reasons:

- the enum member `Red` is automatically assigned the value zero (since it has no initializer and is the first enum member);
- the enum member `Green` is explicitly given the value `10`;
- and the enum member `Blue` is automatically assigned the value one greater than the member that textually precedes it.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

The example

```
enum Circular
{
    A = B,
    B
}
```

results in a compile-time error because the declarations of `A` and `B` are circular. `A` depends on `B` explicitly, and `B` depends on `A` implicitly.

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility -- an enum member is accessible if its containing enum type is accessible.

The System.Enum type

The type `System.Enum` is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from `System.Enum` are available in any enum type. A boxing conversion (Boxing conversions) exists from any enum type to `System.Enum`, and an unboxing conversion (Unboxing conversions) exists from `System.Enum` to any enum type.

Note that `System.Enum` is not itself an *enum_type*. Rather, it is a *class_type* from which all *enum_types* are derived. The type `System.Enum` inherits from the type `System.ValueType` (The System.ValueType type), which, in turn, inherits from type `object`. At run-time, a value of type `System.Enum` can be `null` or a reference to a boxed value of any enum type.

Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion (Explicit enumeration conversions) 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 Enum members). 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: `==`, `!=`, `<`, `>`, `<=`, `>=` (Enumeration comparison operators), binary `+` (Addition operator), binary `-` (Subtraction operator), `^`, `&`, `|` (Enumeration logical operators), `~` (Bitwise complement operator), `++` and `--` (Postfix increment and decrement operators and Prefix increment and decrement operators).

Every enum type automatically derives from the class `System.Enum` (which, in turn, derives from `System.ValueType` and `object`). Thus, inherited methods and properties of this class can be used on values of an enum type.

Delegates

1/26/2018 • 9 minutes to read • Edit Online

Delegates enable scenarios that other languages—such as C++, Pascal, and Modula -- have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate's callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (Delegate declarations) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

Delegate declarations

A *delegate_declaration* is a *type_declaration* (Type declarations) that declares a new delegate type.

```
delegate_declaration
    : attributes? delegate_modifier* 'delegate' return_type
      identifier variant_type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;

delegate_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | delegate_modifier_unsafe
    ;
```

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 The new modifier.

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 (Declared accessibility).

The delegate's type name is *identifier*.

The optional *formal_parameter_list* specifies the parameters of the delegate, and *return_type* indicates the return type of the delegate.

The optional *variant_type_parameter_list* (Variant type parameter lists) specifies the type parameters to the delegate itself.

The return type of a delegate type must be either `void`, or output-safe (Variance safety).

All the formal parameter types of a delegate type must be input-safe. Additionally, any `out` or `ref` parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

Delegate types in C# are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. However, instances of two distinct but structurally equivalent delegate types may compare as equal (Delegate equality operators).

For example:

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

The methods `A.M1` and `B.M1` are compatible with both the delegate types `D1` and `D2`, since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The methods `B.M2`, `B.M3`, and `B.M4` are incompatible with the delegate types `D1` and `D2`, since they have different return types or parameter lists.

Like other generic type declarations, type arguments must be given to create a constructed delegate type. The parameter types and return type of a constructed delegate type are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used in determining what methods are compatible with a constructed delegate type. For example:

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

The method `X.F` is compatible with the delegate type `Predicate<int>` and the method `X.G` is compatible with the delegate type `Predicate<string>`.

The only way to declare a delegate type is via a *delegate_declaration*. A delegate type is a class type that is derived from `System.Delegate`. Delegate types are implicitly `sealed`, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from `System.Delegate`. Note that `System.Delegate` is not itself a delegate type; it is a class type from which all delegate types are derived.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the `System.Delegate` type via the usual member access syntax.

The set of methods encapsulated by a delegate instance is called an invocation list. When a delegate instance is

created (Delegate compatibility) 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 `+` (Addition operator) and `+=` operators (Compound assignment). A delegate can be removed from a combination of delegates, using the binary `-` (Subtraction operator) and `-=` operators (Compound assignment). Delegates can be compared for equality (Delegate equality operators).

The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}

}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // M1
        D cd2 = new D(C.M2);      // M2
        D cd3 = cd1 + cd2;        // M1 + M2
        D cd4 = cd3 + cd1;        // M1 + M2 + M1
        D cd5 = cd4 + cd3;        // M1 + M2 + M1 + M1 + M2
    }

}
```

When `cd1` and `cd2` are instantiated, they each encapsulate one method. When `cd3` is instantiated, it has an invocation list of two methods, `M1` and `M2`, in that order. `cd4`'s invocation list contains `M1`, `M2`, and `M1`, in that order. Finally, `cd5`'s invocation list contains `M1`, `M2`, `M1`, `M1`, and `M2`, in that order. For more examples of combining (as well as removing) delegates, see Delegate invocation.

Delegate compatibility

A method or delegate `M` is ***compatible*** with a delegate type `D` if all of the following are true:

- `D` and `M` have the same number of parameters, and each parameter in `D` has the same `ref` or `out` modifiers as the corresponding parameter in `M`.
- For each value parameter (a parameter with no `ref` or `out` modifier), an identity conversion (Identity conversion) or implicit reference conversion (Implicit reference conversions) exists from the parameter type in `D` to the corresponding parameter type in `M`.
- For each `ref` or `out` parameter, the parameter type in `D` is the same as the parameter type in `M`.
- An identity or implicit reference conversion exists from the return type of `M` to the return type of `D`.

Delegate instantiation

An instance of a delegate is created by a *delegate_creation_expression* (Delegate creation expressions) or a conversion to a delegate type. The newly created delegate instance then refers to either:

- The static method referenced in the *delegate_creation_expression*, or
- The target object (which cannot be `null`) and instance method referenced in the *delegate_creation_expression*, or

- Another delegate.

For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);        // static method
        C t = new C();
        D cd2 = new D(t.M2);        // instance method
        D cd3 = new D(cd2);        // another delegate
    }
}
```

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

Delegate invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. (See Delegate invocations 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 (Reference parameters), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

Attempting to invoke a delegate instance whose value is null results in an exception of type `System.NullReferenceException`.

The following example shows how to instantiate, combine, remove, and invoke delegates:

```
using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);                // call M1

        D cd2 = new D(C.M2);
        cd2(-2);                // call M2

        D cd3 = cd1 + cd2;
        cd3(10);                // call M1 then M2

        cd3 += cd1;
        cd3(20);                // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);                // call M1, M2, M1, then M3

        cd3 -= cd1;             // remove last M1
        cd3(40);                // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);                // call M1 then M2

        cd3 -= cd2;
        cd3(60);                // call M1

        cd3 -= cd2;             // impossible removal is benign
        cd3(60);                // call M1

        cd3 -= cd1;             // invocation list is empty so cd3 is null

        cd3(70);                // System.NullReferenceException thrown

        cd3 -= cd1;             // impossible removal is benign
    }
}
```

As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement, `cd3 -= cd1;`, the delegate `cd3` refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from a

non-empty list) is not an error.

The output produced is:

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

Exceptions

1/13/2018 • 4 minutes to read • Edit Online

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences:

- In C#, all exceptions must be represented by an instance of a class type derived from `System.Exception`. In C++, any value of any type can be used to represent an exception.
- In C#, a finally block (The try statement) can be used to write termination code that executes in both normal execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.
- In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well defined exception classes and are on a par with application-level error conditions.

Causes of exceptions

Exception can be thrown in two different ways.

- A `throw` statement (The throw statement) 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 (Division operator) throws a `System.DivideByZeroException` if the denominator is zero. See Common Exception Classes for a list of the various exceptions that can occur in this way.

The System.Exception class

The `System.Exception` class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- `Message` is a read-only property of type `string` that contains a human-readable description of the reason for the exception.
- `InnerException` is a read-only property of type `Exception`. If its value is non-null, it refers to the exception that caused the current exception—that is, the current exception was raised in a catch block handling the `InnerException`. Otherwise, its value is null, indicating that this exception was not caused by another exception. The number of exception objects chained together in this manner can be arbitrary.

The value of these properties can be specified in calls to the instance constructor for `System.Exception`.

How exceptions are handled

Exceptions are handled by a `try` statement (The try statement).

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 (Static constructors) or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor. The inner exception of the `System.TypeInitializationException` contains the exception that was originally thrown.
- If the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution, and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type) or if there is no base class destructor, then the exception is discarded.

Common Exception Classes

The following exceptions are thrown by certain C# operations.

`System.ArithmeticException`	A base class for exceptions that occur during arithmetic operations, such as `System.DivideByZeroException` and `System.OverflowException`.
`System.ArrayTypeMismatchException`	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
`System.DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`System.IndexOutOfRangeException`	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
`System.InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
`System.NullReferenceException`	Thrown when a `null` reference is used in a way that causes the referenced object to be required.
`System.OutOfMemoryException`	Thrown when an attempt to allocate memory (via `new`) fails.
`System.OverflowException`	Thrown when an arithmetic operation in a `checked` context overflows.
`System.StackOverflowException`	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
`System.TypeInitializationException`	Thrown when a static constructor throws an exception, and no `catch` clauses exists to catch it.

Attributes

1/13/2018 • 25 minutes to read • Edit Online

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method_modifiers* `public`, `protected`, `internal`, and `private`.

C# enables programmers to invent new kinds of declarative information, called ***attributes***. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

Attributes are defined through the declaration of attribute classes (Attribute classes), which may have positional and named parameters (Positional and named parameters). Attributes are attached to entities in a C# program using attribute specifications (Attribute specification), and can be retrieved at run-time as attribute instances (Attribute instances).

Attribute classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an ***attribute class***. The declaration of an attribute class defines a new kind of ***attribute*** that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

Attribute usage

The attribute `AttributeUsage` (The AttributeUsage attribute) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter (Positional and named parameters) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class_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 {...}
```

`AttributeUsage` has a named parameter (Positional and named parameters) called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true, then that attribute class is a ***multi-use attribute class***, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is false or it is unspecified, then that attribute class is a ***single-use attribute class***, and can be specified at most once on an entity.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

shows a class declaration with two uses of the `Author` attribute.

`AttributeUsage` has another named parameter called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class is false then that attribute is not inherited. If it is unspecified, its default value is true.

An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

```
using System;

class X: Attribute {...}
```

is equivalent to the following:

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

Positional and named parameters

Attribute classes can have ***positional parameters*** and ***named parameters***. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {        // Positional parameter
        ...
    }

    public string Topic {                     // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter, `url`, and one named parameter, `Topic`. Although it is non-static and public, the property `Url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the ***attribute parameter types***, which are:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.

- The type `object`.
- The type `System.Type`.
- An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility (Attribute specification).
- Single-dimensional arrays of the above types.
- A constructor argument or public field which does not have one of these types, cannot be used as a positional or named parameter in an attribute specification.

Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for *type_declaration*s (Type declarations), *class_member_declaration*s (Type parameter constraints), *interface_member_declaration*s (Interface members), *struct_member_declaration*s (Struct members), *enum_member_declaration*s (Enum members), *accessor_declarations* (Accessors), *event_accessor_declarations* (Field-like events), and *formal_parameter_list*s (Method parameters).

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_section+
    ;

global_attribute_section
    : '[' global_attribute_target_specifier attribute_list ']'
    | '[' global_attribute_target_specifier attribute_list ',' ']'
    ;

global_attribute_target_specifier
    : global_attribute_target ':'
    ;

global_attribute_target
    : 'assembly'
    | 'module'
    ;

attributes
    : attribute_section+
    ;

attribute_section
    : '[' attribute_target_specifier? attribute_list ']'
    | '[' attribute_target_specifier? attribute_list ',' ']'
    ;

attribute_target_specifier
    : attribute_target ':'
    ;

attribute_target
    : 'field'
    | 'event'
    | 'method'
    | 'param'
    | 'property'
    | 'return'
```

```
    | 'type'
    ;

attribute_list
    : attribute (',' attribute)*
    ;

attribute
    : attribute_name attribute_arguments?
    ;

attribute_name
    : type_name
    ;

attribute_arguments
    : '(' positional_argument_list? ')'
    | '(' positional_argument_list ',' named_argument_list ')'
    | '(' named_argument_list ')'
    ;

positional_argument_list
    : positional_argument (',' positional_argument)*
    ;

positional_argument
    : attribute_argument_expression
    ;

named_argument_list
    : named_argument (','  named_argument)*
    ;

named_argument
    : identifier '=' attribute_argument_expression
    ;

attribute_argument_expression
    : expression
    ;
```

An attribute consists of an *attribute_name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute_argument_expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute_argument_expression*, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

The *attribute_name* identifies an attribute class. If the form of *attribute_name* is *type_name* then this name must refer to an attribute class. Otherwise, a compile-time error occurs. The example

```
class Class1 {}

[Class1] class Class2 {}    // Error
```

results in a compile-time error because it attempts to use `Class1` as an attribute class when `Class1` is not an attribute class.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute_target_specifier*. When an attribute is placed at the global level, a *global_attribute_target_specifier* is required. In all other locations, a reasonable default is applied, but an *attribute_target_specifier* can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, *attribute_target_specifier*s can be omitted except at

the global level. The potentially ambiguous contexts are resolved as follows:

- An attribute specified at global scope can apply either to the target assembly or the target module. No default exists for this context, so an *attribute_target_specifier* is always required in this context. The presence of the `assembly` *attribute_target_specifier* indicates that the attribute applies to the target assembly; the presence of the `module` *attribute_target_specifier* indicates that the attribute applies to the target module.
- An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the delegate. The presence of the `type` *attribute_target_specifier* indicates that the attribute applies to the delegate; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on a method declaration can apply either to the method being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the operator. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the operator; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an event declaration that omits event accessors can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove methods. In the absence of an *attribute_target_specifier*, the attribute applies to the event. The presence of the `event` *attribute_target_specifier* indicates that the attribute applies to the event; the presence of the `field` *attribute_target_specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the methods.
- An attribute specified on a get accessor declaration for a property or indexer declaration can apply either to the associated method or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on a set accessor for a property or indexer declaration can apply either to the associated method or to its lone implicit parameter. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute_target_specifier* indicates that the attribute applies to the parameter; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an add or remove accessor declaration for an event declaration can apply either to the associated method or to its lone parameter. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute_target_specifier* indicates that the attribute applies to the parameter; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.

In other contexts, inclusion of an *attribute_target_specifier* is permitted but unnecessary. For instance, a class declaration may either include or omit the specifier `type`:

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

It is an error to specify an invalid *attribute_target_specifier*. For instance, the specifier `param` cannot be used on a class declaration:

```
[param: Author("Brian Kernighan")]        // Error
class Class1 {}
```

By convention, attribute classes are named with a suffix of `Attribute`. An *attribute_name* of the form *type_name* may either include or omit this suffix. If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the *attribute_name* is spelled such that its right-most *identifier* is a verbatim identifier (Identifiers), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                     // Error: ambiguity
class Class1 {}

[XAttribute]            // Refers to XAttribute
class Class2 {}

[@X]                    // Refers to X
class Class3 {}

[@XAttribute]           // Refers to XAttribute
class Class4 {}
```

shows two attribute classes named `X` and `XAttribute`. The attribute `[X]` is ambiguous, since it could refer to either `X` or `XAttribute`. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute `[XAttribute]` is not ambiguous (although it would be if there was an attribute class named `XAttributeAttribute`!). If the declaration for class `X` is removed, then both attributes refer to the attribute class named `XAttribute`, as follows:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                     // Refers to XAttribute
class Class1 {}

[XAttribute]            // Refers to XAttribute
class Class2 {}

[@X]                    // Error: no attribute named "X"
class Class3 {}
```

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`.

An expression `E` is an *attribute_argument_expression* if all of the following statements are true:

- The type of `E` is an attribute parameter type (Attribute parameter types).
- 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_expression*s.

For example:

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }

    public Type P2 {
        get {...}
        set {...}
    }

    public object P3 {
        get {...}
        set {...}
    }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

A *typeof_expression* (The typeof operator) used as an attribute argument expression can reference a non-generic type, a closed constructed type, or an unbound generic type, but it cannot reference an open type. This is to ensure that the expression can be resolved at compile-time.

```
class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;                  // Error, open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x;        // Ok, closed constructed type
    [A(typeof(List<>))] int y;           // Ok, unbound generic type
}
```

Attribute instances

An ***attribute instance*** is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

Compilation of an attribute

The compilation of an *attribute* with attribute class `T`, *positional_argument_list* `P` and *named_argument_list* `N`, consists of the following steps:

- Follow the compile-time processing steps for compiling an *object_creation_expression* of the form `new T(P)`. These steps either result in a compile-time error, or determine an instance constructor `C` on `T` that can be invoked at run-time.
- If `C` does not have public accessibility, then a compile-time error occurs.
- For each *named_argument* `Arg` in `N`:
  - Let `Name` be the *identifier* of the *named_argument* `Arg`.
  - `Name` must identify a non-static read-write public field or property on `T`. If `T` has no such field or property, then a compile-time error occurs.
- Keep the following information for run-time instantiation of the attribute: the attribute class `T`, the instance constructor `C` on `T`, the *positional_argument_list* `P` and the *named_argument_list* `N`.

Run-time retrieval of an attribute instance

Compilation of an *attribute* yields an attribute class `T`, an instance constructor `C` on `T`, a *positional_argument_list* `P`, and a *named_argument_list* `N`. Given this information, an attribute instance can be retrieved at run-time using the following steps:

- Follow the run-time processing steps for executing an *object_creation_expression* of the form `new T(P)`, using the instance constructor `C` as determined at compile-time. These steps either result in an exception, or produce an instance `O` of `T`.
- For each *named_argument* `Arg` in `N`, in order:
  - Let `Name` be the *identifier* of the *named_argument* `Arg`. If `Name` does not identify a non-static public read-write field or property on `O`, then an exception is thrown.
  - Let `Value` be the result of evaluating the *attribute_argument_expression* of `Arg`.
  - If `Name` identifies a field on `O`, then set this field to `Value`.
  - Otherwise, `Name` identifies a property on `O`. Set this property to `Value`.

- The result is `O`, an instance of the attribute class `T` that has been initialized with the *positional_argument_list* `P` and the *named_argument_list* `N`.

Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

- `System.AttributeUsageAttribute` (The AttributeUsage attribute), which is used to describe the ways in which an attribute class can be used.
- `System.Diagnostics.ConditionalAttribute` (The Conditional attribute), which is used to define conditional methods.
- `System.ObsoleteAttribute` (The Obsolete attribute), which is used to mark a member as obsolete.
- `System.Runtime.CompilerServices.CallerLineNumberAttribute`, `System.Runtime.CompilerServices.CallerFilePathAttribute` and `System.Runtime.CompilerServices.CallerMemberNameAttribute` (Caller info attributes), which are used to supply information about the calling context to optional parameters.

The AttributeUsage attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either directly or indirectly. Otherwise, a compile-time error occurs.

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly     = 0x0001,
        Module       = 0x0002,
        Class        = 0x0004,
        Struct       = 0x0008,
        Enum         = 0x0010,
        Constructor  = 0x0020,
        Method       = 0x0040,
        Property     = 0x0080,
        Field        = 0x0100,
        Event        = 0x0200,
        Interface    = 0x0400,
        Parameter    = 0x0800,
        Delegate     = 0x1000,
        ReturnValue  = 0x2000,

        All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
    }
}
```

The Conditional attribute

The attribute `Conditional` enables the definition of ***conditional methods*** and ***conditional attribute classes***.

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
    }
}
```

Conditional methods

A method decorated with the `Conditional` attribute is a conditional method. The `Conditional` attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the receiver and parameters of the call) is omitted.

A conditional method is subject to the following restrictions:

- The conditional method must be a method in a *class_declaration* or *struct_declaration*. A compile-time error occurs if the `Conditional` attribute is specified on a method in an interface declaration.
- The conditional method must have a return type of `void`.
- The conditional method must not be marked with the `override` modifier. A conditional method may be marked with the `virtual` modifier, however. Overrides of such a method are implicitly conditional, and must not be explicitly marked with a `Conditional` attribute.
- The conditional method must not be an implementation of an interface method. Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional method is used in a *delegate_creation_expression*. The example

```
#define DEBUG

using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`.

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. In the example

File `class1.cs`:

```
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

File `class2.cs`:

```
#define DEBUG

class Class2
{
    public static void G() {
        Class1.F();               // F is called
    }
}
```

File `class3.cs`:

```
#undef DEBUG

class Class3
{
    public static void H() {
        Class1.F();               // F is not called
    }
}
```

the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to `F` in `Class3` is omitted.

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through `base`, of the form `base.M`, are subject to the normal conditional method call rules. In the example

File `class1.cs`:

```
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

File `class2.cs`:

```
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();                       // base.M is not called!
    }
}
```

File `class3.cs`:

```
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                           // M is called
    }
}
```

`Class2` includes a call to the `M` defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol `DEBUG`, which is undefined. Thus, the method writes to the console "`Class2.M executed`" only. Judicious use of *pp_declaration*s can eliminate such problems.

Conditional attribute classes

An attribute class (Attribute classes) decorated with one or more `Conditional` attributes is a ***conditional attribute class***. A conditional attribute class is thus associated with the conditional compilation symbols declared in its `Conditional` attributes. This example:

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declares `TestAttribute` as a conditional attribute class associated with the conditional compilations symbols `ALPHA` and `BETA`.

Attribute specifications (Attribute specification) of a conditional attribute are included if one or more of its associated conditional compilation symbols is defined at the point of specification, otherwise the attribute specification is omitted.

It is important to note that the inclusion or exclusion of an attribute specification of a conditional attribute class is controlled by the conditional compilation symbols at the point of the specification. In the example

File `test.cs`:

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}
```

File `class1.cs`:

```
#define DEBUG

[Test]               // TestAttribute is specified

class Class1 {}
```

File `class2.cs`:

```
#undef DEBUG

[Test]                // TestAttribute is not specified

class Class2 {}
```

the classes `Class1` and `Class2` are each decorated with attribute `Test`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class1` but not `Class2`, the specification of the `Test` attribute on `Class1` is included, while the specification of the `Test` attribute on `Class2` is omitted.

The Obsolete attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}
```

If a program uses a type or member that is decorated with the `Obsolete` attribute, the compiler issues a warning or an error. Specifically, the compiler issues a warning if no error parameter is provided, or if the error parameter is provided and has the value `false`. The compiler issues an error if the error parameter is specified and has the value `true`.

In the example

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
        A a = new A();         // Warning
        a.F();
    }
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `Main` results in a warning that includes the specified message, "This class is obsolete; use class B instead."

Caller info attributes

For purposes such as logging and reporting, it is sometimes useful for a function member to obtain certain compile-time information about the calling code. The caller info attributes provide a way to pass such information transparently.

When an optional parameter is annotated with one of the caller info attributes, omitting the corresponding argument in a call does not necessarily cause the default parameter value to be substituted. Instead, if the specified information about the calling context is available, that information will be passed as the argument value.

For example:

```
using System.Runtime.CompilerServices

...

public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath]   string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line "+ line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}
```

A call to `Log()` with no arguments would print the line number and file path of the call, as well as the name of the member within which the call occurred.

Caller info attributes can occur on optional parameters anywhere, including in delegate declarations. However, the specific caller info attributes have restrictions on the types of the parameters they can attribute, so that there will always be an implicit conversion from a substituted value to the parameter type.

It is an error to have the same caller info attribute on a parameter of both the defining and implementing part of a partial method declaration. Only caller info attributes in the defining part are applied, whereas caller info attributes occurring only in the implementing part are ignored.

Caller information does not affect overload resolution. As the attributed optional parameters are still omitted from the source code of the caller, overload resolution ignores those parameters in the same way it ignores other omitted optional parameters (Overload resolution).

Caller information is only substituted when a function is explicitly invoked in source code. Implicit invocations such as implicit parent constructor calls do not have a source location and will not substitute caller information. Also, calls that are dynamically bound will not substitute caller information. When a caller info attributed parameter is omitted in such cases, the specified default value of the parameter is used instead.

One exception is query-expressions. These are considered syntactic expansions, and if the calls they expand to omit optional parameters with caller info attributes, caller information will be substituted. The location used is the location of the query clause which the call was generated from.

If more than one caller info attribute is specified on a given parameter, they are preferred in the following order: `CallerLineNumber`, `CallerFilePath`, `CallerMemberName`.

The CallerLineNumber attribute

The `System.Runtime.CompilerServices.CallerLineNumberAttribute` is allowed on optional parameters when there is a standard implicit conversion (Standard implicit conversions) from the constant value `int.MaxValue` to the parameter's type. This ensures that any non-negative line number up to that value can be passed without error.

If a function invocation from a location in source code omits an optional parameter with the `CallerLineNumberAttribute`, then a numeric literal representing that location's line number is used as an argument to the invocation instead of the default parameter value.

If the invocation spans multiple lines, the line chosen is implementation-dependent.

Note that the line number may be affected by `#line` directives (Line directives).

The CallerFilePath attribute

The `System.Runtime.CompilerServices.CallerFilePathAttribute` is allowed on optional parameters when there is a standard implicit conversion (Standard implicit conversions) from `string` to the parameter's type.

If a function invocation from a location in source code omits an optional parameter with the `CallerFilePathAttribute`, then a string literal representing that location's file path is used as an argument to the invocation instead of the default parameter value.

The format of the file path is implementation-dependent.

Note that the file path may be affected by `#line` directives (Line directives).

The CallerMemberName attribute

The `System.Runtime.CompilerServices.CallerMemberNameAttribute` is allowed on optional parameters when there is a standard implicit conversion (Standard implicit conversions) from `string` to the parameter's type.

If a function invocation from a location within the body of a function member or within an attribute applied to the function member itself or its return type, parameters or type parameters in source code omits an optional parameter with the `CallerMemberNameAttribute`, then a string literal representing the name of that member is used as an argument to the invocation instead of the default parameter value.

For invocations that occur within generic methods, only the method name itself is used, without the type parameter list.

For invocations that occur within explicit interface member implementations, only the method name itself is used, without the preceding interface qualification.

For invocations that occur within property or event accessors, the member name used is that of the property or event itself.

For invocations that occur within indexer accessors, the member name used is that supplied by an `IndexerNameAttribute` (The IndexerName attribute) on the indexer member, if present, or the default name `Item` otherwise.

For invocations that occur within declarations of instance constructors, static constructors, destructors and operators the member name used is implementation-dependent.

Attributes for Interoperation

Note: This section is applicable only to the Microsoft .NET implementation of C#.

Interoperation with COM and Win32 components

The .NET run-time provides a large number of attributes that enable C# programs to interoperate with components written using COM and Win32 DLLs. For example, the `DllImport` attribute can be used on a `static extern` method to indicate that the implementation of the method is to be found in a Win32 DLL. These attributes are found in the `System.Runtime.InteropServices` namespace, and detailed documentation for these attributes is found in the .NET runtime documentation.

Interoperation with other .NET languages

The IndexerName attribute

Indexers are implemented in .NET using indexed properties, and have a name in the .NET metadata. If no `IndexerName` attribute is present for an indexer, then the name `Item` is used by default. The `IndexerName` attribute enables a developer to override this default and specify a different name.

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

Unsafe code

1/26/2018 • 39 minutes to read • Edit Online

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.

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 productions are shown below.

```
class_modifier_unsafe
    : 'unsafe'
    ;

struct_modifier_unsafe
    : 'unsafe'
    ;

interface_modifier_unsafe
    : 'unsafe'
    ;

delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
    : 'unsafe' block
    ;
```

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.

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. 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`.

Pointer types

In an unsafe context, a *type* (Types) may be a *pointer_type* as well as a *value_type* or a *reference_type*. However, a *pointer_type* may also be used in a `typeof` expression (Anonymous object creation expressions) outside of an unsafe context as such usage is not unsafe.

```
type_unsafe
    : 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* or constructed type, and doesn't contain *reference_type* or constructed 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 is not a constructed type and 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.

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`

EXAMPLE	DESCRIPTION
`void*`	Pointer to unknown type

For a given implementation, all pointer types must have the same size and representation.

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;
```

The value of a pointer having type `T*` represents the address of a variable of type `T`. The pointer indirection operator `*` (Pointer indirection) 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 (Boxing and unboxing) 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 Pointer conversions.

A *pointer_type* cannot be used as a type argument (Constructed types), and type inference (Type inference) fails on generic method calls that would have inferred a type argument to be a pointer type.

A *pointer_type* may be used as the type of a volatile field (Volatile fields).

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
        }
    }
}
```

A method can return a value of some type, and that type can be a pointer. 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;
}
```

In an unsafe context, several constructs are available for operating on pointers:

- The `*` operator may be used to perform pointer indirection (Pointer indirection).
- The `->` operator may be used to access a member of a struct through a pointer (Pointer member access).
- The `[]` operator may be used to index a pointer (Pointer element access).
- The `&` operator may be used to obtain the address of a variable (The address-of operator).
- The `++` and `--` operators may be used to increment and decrement pointers (Pointer increment and decrement).
- The `+` and `-` operators may be used to perform pointer arithmetic (Pointer arithmetic).
- The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (Pointer comparison).
- The `stackalloc` operator may be used to allocate memory from the call stack (Fixed size buffers).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (The fixed statement).

Fixed and moveable variables

The address-of operator (The address-of operator) and the `fixed` statement (The fixed statement) 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 (The address-of operator) 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 (The fixed statement), 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* (Simple names) that refers to a local variable or a value parameter, unless the variable is captured by an anonymous function.
- A variable resulting from a *member_access* (Member access) of the form `V.I`, where `V` is a fixed variable of a *struct_type*.
- A variable resulting from a *pointer_indirection_expression* (Pointer indirection) of the form `*P`, a *pointer_member_access* (Pointer member access) of the form `P->I`, or a *pointer_element_access* (Pointer element access) 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.

Pointer conversions

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- From any *pointer_type* to the type `void*`.
- From the `null` literal to any *pointer_type*.

Additionally, in an unsafe context, the set of available explicit conversions (Explicit conversions) 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 (Standard implicit conversions) 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`.

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
```

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. For example, the following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;

class Test
{
    unsafe static void Main() {
      double d = 123.456e23;
        unsafe {
           byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
               Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

Of course, the output produced depends on endianness.

Mappings between pointers and integers are implementation-defined. 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.

Pointer arrays

In an unsafe context, arrays of pointers can be constructed. Only some of the conversions that apply to other array types are allowed on pointer arrays:

- The implicit reference conversion (Implicit reference conversions) from any *array_type* to `System.Array` and the interfaces it implements also applies to pointer arrays. However, any attempt to access the array elements through `System.Array` or the interfaces it implements will result in an exception at run-time, as pointer types are not convertible to `object`.
- The implicit and explicit reference conversions (Implicit reference conversions, Explicit reference conversions) from a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its generic base interfaces never apply to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.
- The explicit reference conversion (Explicit reference conversions) from `System.Array` and the interfaces it implements to any *array_type* applies to pointer arrays.
- The explicit reference conversions (Explicit reference conversions) from `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]` never applies to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.

These restrictions mean that the expansion for the `foreach` statement over arrays described in The foreach statement cannot be applied to pointer arrays. Instead, a foreach statement of the form

```
foreach (V v in x) embedded_statement
```

where the type of `x` is an array type of the form `T[,,...,]`, `N` is the number of dimensions minus 1 and `T` or `V` is a pointer type, is expanded using nested for-loops as follows:

```
{
    T[,,...,] a = x;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
        V v = (V)a.GetValue(i0,i1,...,iN);
        embedded_statement
    }
}
```

The variables `a`, `i0`, `i1`, ..., `iN` are not visible to or accessible to `x` or the *embedded_statement* or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (Pointer conversions) from `T` (the element type) to `V`, an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at run-time.

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* (Simple names), *member_access* (Member access), *invocation_expression* (Invocation expressions), or *element_access* (Element access) is of a pointer type.

In an unsafe context, the *primary_no_array_creation_expression* (Primary expressions) and *unary_expression* (Unary operators) productions permit the following additional constructs:

```
primary_no_array_creation_expression_unsafe
    : pointer_member_access
    | pointer_element_access
    | sizeof_expression
    ;

unary_expression_unsafe
    : pointer_indirection_expression
    | addressof_expression
    ;
```

These constructs are described in the following sections. The precedence and associativity of the unsafe operators is implied by the grammar.

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. 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 Pointer conversions), and the address of a variable after the end of its lifetime.

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned (Initially assigned variables).

Pointer member access

A *pointer_member_access* consists of a *primary_expression*, followed by a "`->`" token, followed by an *identifier* and an optional *type_argument_list*.

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

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 Pointer indirection. For a description of the member access operator (`.`), see Member access.

In the example

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

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 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 Pointer indirection. For a description of the pointer addition operator (`+`), see Pointer arithmetic.

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;
        }
    }
}
```

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. This is the same as C and C++.

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 (Fixed and moveable variables), the construct `&E` computes the address of the variable given by `E`. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a read-only local variable, or if `E` denotes a moveable variable. In the last case, a fixed statement (The fixed statement) can be used to temporarily "fix" the variable before obtaining its address. As stated in Member access, 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.

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.

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.

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.

Pointer increment and decrement

In an unsafe context, the `++` and `--` operators (Postfix increment and decrement operators and Prefix increment and decrement operators) 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 (Pointer arithmetic). 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.

Pointer arithmetic

In an unsafe context, the `+` and `-` operators (Addition operator and Subtraction operator) 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)`.

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
```

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.

Pointer comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (Relational and type-testing operators) 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.

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* (Pointer types).

```
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.

The fixed statement

In an unsafe context, the *embedded_statement* (Statements) 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.

```
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')' embedded_statement
    ;

fixed_pointer_declarators
    : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;

fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;

fixed_pointer_initializer
    : '&' variable_reference
    | expression
    ;
```

Each *fixed_pointer_declarator* declares a local variable of the given *pointer_type* and initializes that local variable with the address computed by the corresponding *fixed_pointer_initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed_pointer_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* (Precise rules for determining definite assignment) to a moveable variable (Fixed and moveable variables) 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. If the array expression is null or if the array has zero elements, the initializer computes an address equal to zero.
- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.

- A *simple_name* or *member_access* that references a fixed size buffer member of a moveable variable, provided the type of the fixed size buffer member is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes a pointer to the first element of the fixed size buffer (Fixed size buffers in expressions), and the fixed size buffer is guaranteed to remain at a fixed address for the duration of the `fixed` statement.

For each address computed by a *fixed_pointer_initializer* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. 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.

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 fourth `fixed` statement in the example above produces a similar result to the third.

This example of the `fixed` statement uses `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);
        }
    }
}
```

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.

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
```

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.

In the example:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}
```

a fixed statement is used to fix a fixed size buffer of a struct so its address can be used as a pointer.

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 results in undefined behavior. 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.

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*`.

Fixed size buffers

Fixed size buffers are used to declare "C style" in-line arrays as members of structs, and are primarily useful for interfacing with unmanaged APIs.

Fixed size buffer declarations

A ***fixed size buffer*** is a member that represents storage for a fixed length buffer of variables of a given type. A fixed size buffer declaration introduces one or more fixed size buffers of a given element type. Fixed size buffers are only permitted in struct declarations and can only occur in unsafe contexts (Unsafe contexts).

```
struct_member_declaration_unsafe
    : fixed_size_buffer_declaration
    ;

fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type fixed_size_buffer_declarator+ ';'
    ;

fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;

buffer_element_type
    : type
    ;

fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

A fixed size buffer declaration may include a set of attributes (Attributes), a `new` modifier (Modifiers), a valid combination of the four access modifiers (Type parameters and constraints) and an `unsafe` modifier (Unsafe contexts). The attributes and modifiers apply to all of the members declared by the fixed size buffer declaration. It is an error for the same modifier to appear multiple times in a fixed size buffer declaration.

A fixed size buffer declaration is not permitted to include the `static` modifier.

The buffer element type of a fixed size buffer declaration specifies the element type of the buffer(s) introduced by the declaration. The buffer element type must be one of the predefined types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `bool`.

The buffer element type is followed by a list of fixed size buffer declarators, each of which introduces a new member. A fixed size buffer declarator consists of an identifier that names the member, followed by a constant expression enclosed in `[` and `]` tokens. The constant expression denotes the number of elements in the member introduced by that fixed size buffer declarator. The type of the constant expression must be implicitly convertible to type `int`, and the value must be a non-zero positive integer.

The elements of a fixed size buffer are guaranteed to be laid out sequentially in memory.

A fixed size buffer declaration that declares multiple fixed size buffers is equivalent to multiple declarations of a single fixed size buffer declaration with the same attributes, and element types. For example

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
   public fixed int z[100];
}
```

Fixed size buffers in expressions

Member lookup (Operators) of a fixed size buffer member proceeds exactly like member lookup of a field.

A fixed size buffer can be referenced in an expression using a *simple_name* (Type inference) or a *member_access* (Compile-time checking of dynamic overload resolution).

When a fixed size buffer member is referenced as a simple name, the effect is the same as a member access of the form `this.I`, where `I` is the fixed size buffer member.

In a member access of the form `E.I`, if `E` is of a struct type and a member lookup of `I` in that struct type identifies a fixed size member, then `E.I` is evaluated an classified as follows:

- If the expression `E.I` does not occur in an unsafe context, a compile-time error occurs.
- If `E` is classified as a value, a compile-time error occurs.
- Otherwise, if `E` is a moveable variable (Fixed and moveable variables) and the expression `E.I` is not a *fixed_pointer_initializer* (The fixed statement), a compile-time error occurs.
- Otherwise, `E` references a fixed variable and the result of the expression is a pointer to the first element of the fixed size buffer member `I` in `E`. The result is of type `S*`, where `S` is the element type of `I`, and is classified as a value.

The subsequent elements of the fixed size buffer can be accessed using pointer operations from the first element. Unlike access to arrays, access to the elements of a fixed size buffer is an unsafe operation and is not range checked.

The following example declares and uses a struct with a fixed size buffer member.

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

Definite assignment checking

Fixed size buffers are not subject to definite assignment checking (Definite assignment), and fixed size buffer members are ignored for purposes of definite assignment checking of struct type variables.

When the outermost containing struct variable of a fixed size buffer member is a static variable, an instance variable of a class instance, or an array element, the elements of the fixed size buffer are automatically initialized to their default values (Default values). In all other cases, the initial content of a fixed size buffer is undefined.

Stack allocation

In an unsafe context, a local variable declaration (Local variable declarations) may include a stack allocation initializer which allocates memory from the call stack.

```
local_variable_initializer_unsafe
    : 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 (Pointer types) 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 (The try statement).

There is no way to explicitly free memory allocated using `stackalloc`. All stack allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. This corresponds to the `alloca` function, an extension commonly found in C and C++ implementations.

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.

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. 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;
```

```
            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);
            try {
                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);
            }
            finally {
                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.

Documentation comments

1/13/2018 • 22 minutes to read • Edit Online

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. In source code files, comments having a certain form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Comments using such syntax are called ***documentation comments***. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method). The XML generation tool is called the ***documentation generator***. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the ***documentation file***. A documentation file is used as input to a ***documentation viewer***; a tool intended to produce some sort of visual display of type information and its associated documentation.

This specification suggests a set of tags to be used in documentation comments, but use of these tags is not required, and other tags may be used if desired, as long the rules of well-formed XML are followed.

Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are single-line comments that start with three slashes (`///`), or delimited comments that start with a slash and two stars (`/**`). They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate. Attribute sections (Attribute specification) are considered part of declarations, so documentation comments must precede attributes applied to a type or member.

Syntax:

```
single_line_doc_comment
    : '///' input_character*
    ;

delimited_doc_comment
    : '/**' delimited_comment_section* asterisk+ '/'
    ;
```

In a *single_line_doc_comment*, if there is a *whitespace* character following the `///` characters on each of the *single_line_doc_comment*s 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 line 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:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
    /// <summary>method <c>draw</c> renders the point.</summary>
    void draw() {...}
}
```

The text within documentation comments must be well formed according to the rules of XML (http://www.w3.org/TR/REC-xml). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in Recommended tags. Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a `cref` attribute, the documentation generator must respect namespace visibility according to `using` statements appearing within the source code. For code elements that are generic, the normal generic syntax (ie "`List<T>`") cannot be used because it produces invalid XML. Braces can be used instead of brackets (ie "`List{T}`"), or the XML escape syntax can be used (ie "`List<T>`").
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.
- The `<include>` tag includes information from an external XML file.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

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	SECTION	PURPOSE
`<c>`	`<c>`	Set text in a code-like font
`<code>`	`<code>`	Set one or more lines of source code or program output
`<example>`	`<example>`	Indicate an example
`<exception>`	`<exception>`	Identifies the exceptions a method can throw
`<include>`	`<include>`	Includes XML from an external file
`<list>`	`<list>`	Create a list or table

TAG	SECTION	PURPOSE
`<para>`	`<para>`	Permit structure to be added to text
`<param>`	`<param>`	Describe a parameter for a method or constructor
`<paramref>`	`<paramref>`	Identify that a word is a parameter name
`<permission>`	`<permission>`	Document the security accessibility of a member
`<remark>`	`<remark>`	Describe additional information about a type
`<returns>`	`<returns>`	Describe the return value of a method
`<see>`	`<see>`	Specify a link
`<seealso>`	`<seealso>`	Generate a See Also entry
`<summary>`	`<summary>`	Describe a type or a member of a type
`<value>`	`<value>`	Describe a property
`<typeparam>`		Describe a generic type parameter
`<typeparamref>`		Identify that a word is a type parameter name

`<c>`

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use `<code>` (`<code>`).

Syntax:

```
<c>text</c>
```

Example:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>

public class Point
{
    // ...
}
```

`<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>` (`<c>`).

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;
}
```

`<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>` (`<code>`) as well.

Syntax:

```
<example>description</example>
```

Example:

See `<code>` (`<code>`) for an example.

`<exception>`

This tag provides a way to document the exceptions a method can throw.

Syntax:

```
<exception cref="member">description</exception>
```

where

- `member` is 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` is 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();
        // ...
    }
}
```

`<include>`

This tag allows including information from an XML document that is external to the source code file. The external file must be a well-formed XML document, and an XPath expression is applied to that document to specify what XML from that document to include. The `<include>` tag is then replaced with the selected XML from the external document.

Syntax:

```
<include file="filename" path="xpath" />
```

where

- `filename` is the file name of an external XML file. The file name is interpreted relative to the file that contains the include tag.
- `xpath` is an XPath expression that selects some of the XML in the external XML file.

Example:

If the source code contained a declaration like:

```
/// <include file="docs.xml" *path=*'extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

and the external file "docs.xml" had the following contents:

```
<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
  <class name="StringList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
</extradoc>
```

then the same documentation is output as if the source code contained:

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { ... }
```

`<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` is the term to define, whose definition is in `description`.
- `description` is either an item in a bullet or numbered list, or the definition of a `term`.

Example:

```
public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}
```

`<para>`

This tag is for use inside other tags, such as `<summary>` (`<remark>`) or `<returns>` (`<returns>`), and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

where `content` is 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-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

`<param>`

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

- `name` is the name of the parameter.
- `description` is a description of the parameter.

Example:

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<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` is the name of the parameter.

Example:

```
/// <summary>This constructor initializes the new Point to
///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<permission>`

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

- `member` is 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` is a description of the access to the member.

Example:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>

public static void Test() {
    // ...
}
```

`<remark>`

This tag is used to specify extra information about a type. (Use `<summary>` (`<summary>`) to describe the type itself and the members of a type.)

Syntax:

```
<remark>description</remark>
```

where `description` is the text of the remark.

Example:

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remark>Uses polar coordinates</remark>
public class Point
{
    // ...
}
```

`<returns>`

This tag is used to describe the return value of a method.

Syntax:

```
<returns>description</returns>
```

where `description` is a description of the return value.

Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///    without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + "," + Y + ")";
}
```

`<see>`

This tag allows a link to be specified within text. Use `<seealso>` (`<seealso>`) to indicate text that is to appear in a See Also section.

Syntax:

```
<see cref="member"/>
```

where `member` is the name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method changes the point's location to
///    the given coordinates.</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;
}
```

`<seealso>`

This tag allows an entry to be generated for the See Also section. Use `<see>` (`<see>`) to specify a link from within text.

Syntax:

```
<seealso cref="member"/>
```

where `member` is the name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method determines whether two Points have the same
///    location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

`<summary>`

This tag can be used to describe a type or a member of a type. Use `<remark>` (`<remark>`) to describe the type itself.

Syntax:

```
<summary>description</summary>
```

where `description` is a summary of the type or member.

Example:

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

`<value>`

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where `property description` is 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; }
}
```

`<typeparam>`

This tag is used to describe a generic type parameter for a class, struct, interface, delegate, or method.

Syntax:

```
<typeparam name="name">description</typeparam>
```

where `name` is the name of the type parameter, and `description` is its description.

Example:

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

`<typeparamref>`

This tag is used to indicate that a word is a type parameter. The documentation file can be processed to format this type parameter in some distinct way.

Syntax:

```
<typeparamref name="name"/>
```

where `name` is the name of the type parameter.

Example:

```
/// <summary>This method fetches data and returns a list of <typeparamref name="T"/>.</summary>
/// <param name="query">query to execute</param>
public List<T> FetchData<T>(string query) {
    ...
}
```

Processing the documentation file

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.

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)

CHARACTER	DESCRIPTION
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 `#(U+0023)` characters. (It is assumed that no element has this character in its name.)
- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows:
  - Arguments are represented by their documentation name, which is based on their fully qualified name, modified as follows:
    - Arguments that represent generic types have an appended "`" character followed by the number of type parameters
    - Arguments having the `out` or `ref` modifier have an `@` following their type name. Arguments passed by value or via `params` have no special notation.
    - Arguments that are arrays are represented as `[lowerbound:size, ... , lowerbound:size]` where the number of commas is the rank less one, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the "`:`" is omitted as well. Jagged arrays are represented by one "`[]`" per level.
    - Arguments that have pointer types other than void are represented using a `*` following the type name. A void pointer is represented using a type name of `System.Void`.
    - Arguments that refer to generic type parameters defined on types are encoded using the "`" character followed by the zero-based index of the type parameter.
    - Arguments that use generic type parameters defined in methods use a double-backtick "``" instead of the "`" used for types.
    - Arguments that refer to constructed generic types are encoded using the generic type, followed by "{", followed by a comma-separated list of type arguments, followed by "}".

ID string examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name, augmented with generic information:

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

namespace Acme
{
    interface IProcess {...}

    struct ValueType {...}

    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }

    class MyList<T>
    {
        class Helper<U,V> {...}
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
```

- Fields are represented by their fully qualified name:

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }

        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **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) {...}
    }

    class MyList<T>
    {
        public void Test(T t) { }
    }

    class UseList
    {
        public void Process(MyList<int> list) { }
        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }
}

"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(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- 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_LogicalNot`, `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) {...}
    }
}

"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"
```

An example

C# source code

The following example shows the source code of a `Point` class:

```
namespace Graphics
{

/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
/// </summary>
public class Point
{

    /// <summary>Instance variable <c>x</c> represents the point's
    ///    x-coordinate.</summary>
    private int x;

    /// <summary>Instance variable <c>y</c> represents the point's
    ///    y-coordinate.</summary>
    private int y;

    /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
    public int X
    {
        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-trivial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
        // class test code goes here
    }
}
}
```

Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-dimensional
            plane.
            </summary>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>

        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<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-trivial 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>
```