C#

**Language Specification**

**Version 5.0**

Notice

© 1999-2016 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Other product and company names mentioned herein may be the trademarks of their respective owners.

**Table of Contents**

[1 Introduction](#_Toc00001)

[1.1 Hello world](#_Toc00002)

[1.2 Program structure](#_Toc00003)

[1.3 Types and variables](#_Toc00004)

[1.4 Expressions](#_Toc00005)

[1.5 Statements](#_Toc00006)

[1.6 Classes and objects](#_Toc00007)

[1.7 Structs](#_Toc00026)

[1.8 Arrays](#_Toc00027)

[1.9 Interfaces](#_Toc00028)

[1.10 Enums](#_Toc00029)

[1.11 Delegates](#_Toc00030)

[1.12 Attributes](#_Toc00031)

[2 Lexical structure](#_Toc00032)

[2.1 Programs](#_Toc00033)

[2.2 Grammars](#_Toc00034)

[2.3 Lexical analysis](#_Toc00038)

[2.4 Tokens](#_Toc00042)

[2.5 Pre-processing directives](#_Toc00054)

[3 Basic concepts](#_Toc00064)

[3.1 Application Startup](#_Toc00065)

[3.2 Application termination](#_Toc00066)

[3.3 Declarations](#_Toc00067)

[3.4 Members](#_Toc00068)

[3.5 Member access](#_Toc00076)

[3.6 Signatures and overloading](#_Toc00081)

[3.7 Scopes](#_Toc00082)

[3.8 Namespace and type names](#_Toc00086)

[3.9 Automatic memory management](#_Toc00088)

[3.10 Execution order](#_Toc00089)

[4 Types](#_Toc00090)

[4.1 Value types](#_Toc00091)

[4.2 Reference types](#_Toc00102)

[4.3 Boxing and unboxing](#_Toc00110)

[4.4 Constructed types](#_Toc00113)

[4.5 Type parameters](#_Toc00118)

[4.6 Expression tree types](#_Toc00119)

[5 Variables](#_Toc00120)

[5.1 Variable categories](#_Toc00121)

[5.2 Default values](#_Toc00131)

[5.3 Definite assignment](#_Toc00132)

[5.4 Variable references](#_Toc00165)

[5.5 Atomicity of variable references](#_Toc00166)

[6 Conversions](#_Toc00167)

[6.1 Implicit conversions](#_Toc00168)

[6.2 Explicit conversions](#_Toc00181)

[6.3 Standard conversions](#_Toc00190)

[6.4 User-defined conversions](#_Toc00193)

[6.5 Anonymous function conversions](#_Toc00199)

[6.6 Method group conversions](#_Toc00203)

[7 Expressions](#_Toc00204)

[7.1 Expression classifications](#_Toc00205)

[7.2 Static and Dynamic Binding](#_Toc00207)

[7.3 Operators](#_Toc00211)

[7.4 Member lookup](#_Toc00221)

[7.5 Function members](#_Toc00223)

[7.6 Primary expressions](#_Toc00252)

[7.7 Unary operators](#_Toc00281)

[7.8 Arithmetic operators](#_Toc00292)

[7.9 Shift operators](#_Toc00298)

[7.10 Relational and type-testing operators](#_Toc00299)

[7.11 Logical operators](#_Toc00311)

[7.12 Conditional logical operators](#_Toc00316)

[7.13 The null coalescing operator](#_Toc00319)

[7.14 Conditional operator](#_Toc00320)

[7.15 Anonymous function expressions](#_Toc00321)

[7.16 Query expressions](#_Toc00330)

[7.17 Assignment operators](#_Toc00341)

[7.18 Expression](#_Toc00345)

[7.19 Constant expressions](#_Toc00346)

[7.20 Boolean expressions](#_Toc00347)

[8 Statements](#_Toc00348)

[8.1 End points and reachability](#_Toc00349)

[8.2 Blocks](#_Toc00350)

[8.3 The empty statement](#_Toc00352)

[8.4 Labeled statements](#_Toc00353)

[8.5 Declaration statements](#_Toc00354)

[8.6 Expression statements](#_Toc00357)

[8.7 Selection statements](#_Toc00358)

[8.8 Iteration statements](#_Toc00361)

[8.9 Jump statements](#_Toc00366)

[8.10 The try statement](#_Toc00372)

[8.11 The checked and unchecked statements](#_Toc00373)

[8.12 The lock statement](#_Toc00374)

[8.13 The using statement](#_Toc00375)

[8.14 The yield statement](#_Toc00376)

[9 Namespaces](#_Toc00377)

[9.1 Compilation units](#_Toc00378)

[9.2 Namespace declarations](#_Toc00379)

[9.3 Extern aliases](#_Toc00380)

[9.4 Using directives](#_Toc00381)

[9.5 Namespace members](#_Toc00384)

[9.6 Type declarations](#_Toc00385)

[9.7 Namespace alias qualifiers](#_Toc00386)

[10 Classes](#_Toc00388)

[10.1 Class declarations](#_Toc00389)

[10.2 Partial types](#_Toc00401)

[10.3 Class members](#_Toc00410)

[10.4 Constants](#_Toc00430)

[10.5 Fields](#_Toc00431)

[10.6 Methods](#_Toc00441)

[10.7 Properties](#_Toc00457)

[10.8 Events](#_Toc00463)

[10.9 Indexers](#_Toc00468)

[10.10 Operators](#_Toc00470)

[10.11 Instance constructors](#_Toc00474)

[10.12 Static constructors](#_Toc00481)

[10.13 Destructors](#_Toc00482)

[10.14 Iterators](#_Toc00483)

[10.15 Async functions](#_Toc00494)

[11 Structs](#_Toc00497)

[11.1 Struct declarations](#_Toc00498)

[11.2 Struct members](#_Toc00503)

[11.3 Class and struct differences](#_Toc00504)

[11.4 Struct examples](#_Toc00515)

[12 Arrays](#_Toc00518)

[12.1 Array types](#_Toc00519)

[12.2 Array creation](#_Toc00522)

[12.3 Array element access](#_Toc00523)

[12.4 Array members](#_Toc00524)

[12.5 Array covariance](#_Toc00525)

[12.6 Array initializers](#_Toc00526)

[13 Interfaces](#_Toc00527)

[13.1 Interface declarations](#_Toc00528)

[13.2 Interface members](#_Toc00536)

[13.3 Fully qualified interface member names](#_Toc00542)

[13.4 Interface implementations](#_Toc00543)

[14 Enums](#_Toc00551)

[14.1 Enum declarations](#_Toc00552)

[14.2 Enum modifiers](#_Toc00553)

[14.3 Enum members](#_Toc00554)

[14.4 The System.Enum type](#_Toc00555)

[14.5 Enum values and operations](#_Toc00556)

[15 Delegates](#_Toc00557)

[15.1 Delegate declarations](#_Toc00558)

[15.2 Delegate compatibility](#_Toc00559)

[15.3 Delegate instantiation](#_Toc00560)

[15.4 Delegate invocation](#_Toc00561)

[16 Exceptions](#_Toc00562)

[16.1 Causes of exceptions](#_Toc00563)

[16.2 The System.Exception class](#_Toc00564)

[16.3 How exceptions are handled](#_Toc00565)

[16.4 Common Exception Classes](#_Toc00566)

[17 Attributes](#_Toc00567)

[17.1 Attribute classes](#_Toc00568)

[17.2 Attribute specification](#_Toc00572)

[17.3 Attribute instances](#_Toc00573)

[17.4 Reserved attributes](#_Toc00576)

[17.5 Attributes for Interoperation](#_Toc00586)

[18 Unsafe code](#_Toc00590)

[18.1 Unsafe contexts](#_Toc00591)

[18.2 Pointer types](#_Toc00592)

[18.3 Fixed and moveable variables](#_Toc00593)

[18.4 Pointer conversions](#_Toc00594)

[18.5 Pointers in expressions](#_Toc00596)

[18.6 The fixed statement](#_Toc00605)

[18.7 Fixed size buffers](#_Toc00606)

[18.8 Stack allocation](#_Toc00610)

[18.9 Dynamic memory allocation](#_Toc00611)

[19 Documentation comments](#_Toc00612)

[19.1 Introduction](#_Toc00613)

[19.2 Recommended tags](#_Toc00614)

[19.3 Processing the documentation file](#_Toc00633)

[19.4 An example](#_Toc00636)

# Introduction

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](#_Trm00008) 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](#_Trm00009) consist of one or more source files. Programs declare [types](#_Trm00011), which contain [members](#_Trm00012) and can be organized into [namespaces](#_Trm00010). Classes and interfaces are examples of [types](#_Trm00011). Fields, methods, properties, and events are examples of [members](#_Trm00012). When C# [programs](#_Trm00009) are compiled, they are physically packaged into [assemblies](#_Trm00013). 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](#_Trm00012): a field named top, two methods named Push and Pop, and a nested class named Entry. The Entry class further contains three [members](#_Trm00012): 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](#_Trm00017), there is no need for #include directives and header files in C#. The public [types](#_Trm00011) and [members](#_Trm00012) 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](#_Trm00011) in C#: ***value types*** and ***reference types***. Variables of value [types](#_Trm00011) directly contain their data whereas variables of [reference types](#_Trm00019) store references to their data, the latter being known as objects. With [reference types](#_Trm00019), 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](#_Trm00011), 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](#_Trm00011) are further divided into ***simple types***, ***enum types***, ***struct types***, and ***nullable types***, and C#'s [reference types](#_Trm00019) 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](#_Trm00011) | Simple [types](#_Trm00011) | Signed integral: sbyte, short, int, long |
|  |  | Unsigned integral: byte, ushort, uint, ulong |
|  |  | Unicode characters: char |
|  |  | IEEE floating point: float, double |
|  |  | High-precision decimal: decimal |
|  |  | Boolean: bool |
|  | Enum [types](#_Trm00011) | User-defined [types](#_Trm00011) of the form enum E {...} |
|  | Struct [types](#_Trm00011) | User-defined [types](#_Trm00011) of the form struct S {...} |
|  | Nullable [types](#_Trm00011) | Extensions of all other value [types](#_Trm00011) with a null value |
| Reference [types](#_Trm00011) | Class [types](#_Trm00011) | Ultimate base class of all other [types](#_Trm00011): object |
|  |  | Unicode strings: string |
|  |  | User-defined [types](#_Trm00011) of the form class C {...} |
|  | Interface [types](#_Trm00011) | User-defined [types](#_Trm00011) of the form interface I {...} |
|  | Array [types](#_Trm00011) | Single- and multi-dimensional, for example, int[] and int[,] |
|  | Delegate [types](#_Trm00011) | User-defined [types](#_Trm00011) of the form e.g. delegate int D(...) |

The eight integral [types](#_Trm00011) provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

The two floating point [types](#_Trm00011), 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](#_Trm00011).

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Bits** | **Type** | **Range/Precision** |
| Signed integral | 8 | sbyte | -128...127 |
|  | 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](#_Trm00009) use ***type declarations*** to create new [types](#_Trm00011). A type declaration specifies the name and the [members](#_Trm00012) of the new type. Five of C#'s categories of [types](#_Trm00011) are user-definable: [class types](#_Trm00024), [struct types](#_Trm00022), [interface types](#_Trm00025), [enum types](#_Trm00021), and [delegate types](#_Trm00027).

A class type defines a data structure that contains data [members](#_Trm00012) (fields) and function [members](#_Trm00012) (methods, properties, and others). Class [types](#_Trm00011) 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](#_Trm00012) and function [members](#_Trm00012). However, unlike classes, structs are value [types](#_Trm00011) and do not require heap allocation. Struct [types](#_Trm00011) do not support user-specified inheritance, and all [struct types](#_Trm00022) implicitly inherit from type object.

An interface type defines a contract as a named set of public function [members](#_Trm00012). A class or struct that implements an interface must provide implementations of the interface's function [members](#_Trm00012). 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](#_Trm00006).

Class, struct, interface and [delegate types](#_Trm00027) all support generics, whereby they can be parameterized with other [types](#_Trm00011).

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](#_Trm00011). 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](#_Trm00011) listed above, [array types](#_Trm00026) do not have to be declared before they can be used. Instead, [array types](#_Trm00026) 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](#_Trm00011) 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](#_Trm00011). Values of [reference types](#_Trm00019) are treated as objects simply by viewing the values as type object. Values of value [types](#_Trm00011) 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](#_Trm00007) effectively means that value [types](#_Trm00011) can become objects "on demand." Because of the unification, general-purpose [libraries](#_Trm00015) that use type object can be used with both [reference types](#_Trm00019) and value [types](#_Trm00011).

There are several kinds of ***variables*** in C#, including fields, array elements, local [variables](#_Trm00031), 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 |
| 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](#_Trm00034) of an expression indicate which operations to apply to the [operands](#_Trm00033). Examples of [operators](#_Trm00034) include +, -, \*, /, and new. Examples of [operands](#_Trm00033) include literals, fields, local [variables](#_Trm00031), and expressions.

When an expression contains multiple [operators](#_Trm00034), the ***precedence*** of the [operators](#_Trm00034) controls the order in which the individual [operators](#_Trm00034) are evaluated. For example, the expression x + y \* z is evaluated as x + (y \* z) because the \* operator has higher [precedence](#_Trm00035) than the + operator.

Most [operators](#_Trm00034) can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the [operands](#_Trm00033) are of a user-defined class or struct type.

The following table summarizes C#'s [operators](#_Trm00034), listing the operator categories in order of [precedence](#_Trm00035) from highest to lowest. Operators in the same category have equal [precedence](#_Trm00035).

|  |  |  |
| --- | --- | --- |
| **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 |
|  | 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 |
| 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](#_Trm00034) 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](#_Trm00037), a number of which are defined in terms of embedded [statements](#_Trm00037).

A ***block*** permits multiple [statements](#_Trm00037) to be written in contexts where a single statement is allowed. A [block](#_Trm00038) consists of a list of [statements](#_Trm00037) written between the delimiters { and }.

***Declaration statements*** are used to declare local [variables](#_Trm00031) and constants.

***Expression statements*** are used to evaluate expressions. [Expressions](#_Trm00032) that can be used as [statements](#_Trm00037) include method invocations, object allocations using the new operator, assignments using = and the compound assignment [operators](#_Trm00034), increment and decrement operations using the ++ and -- [operators](#_Trm00034) and await expressions.

***Selection statements*** are used to select one of a number of possible [statements](#_Trm00037) for execution based on the value of some expression. In this group are the if and switch [statements](#_Trm00037).

***Iteration statements*** are used to repeatedly execute an embedded statement. In this group are the while, do, for, and foreach [statements](#_Trm00037).

***Jump statements*** are used to transfer control. In this group are the break, continue, goto, throw, return, and yield [statements](#_Trm00037).

The try...catch statement is used to catch exceptions that occur during execution of a [block](#_Trm00038), 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](#_Trm00037) 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](#_Trm00011). A class is a data structure that combines state (fields) and actions (methods and other function [members](#_Trm00012)) in a single unit. A class provides a definition for dynamically created ***instances*** of the class, also known as ***objects***. [Classes](#_Trm00044) 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](#_Trm00037) create two Point [objects](#_Trm00046) and store references to those [objects](#_Trm00046) in two [variables](#_Trm00031):

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](#_Trm00046) in C#.

### Members

The [members](#_Trm00012) of a class are either ***static members*** or ***instance members***. Static [members](#_Trm00012) belong to classes, and [instance members](#_Trm00052) belong to [objects](#_Trm00046) ([instances](#_Trm00045) of classes).

The following table provides an overview of the kinds of [members](#_Trm00012) 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](#_Trm00045) of the class like an array |
| Events | Notifications that can be generated by the class |
| Operators | Conversions and expression [operators](#_Trm00034) supported by the class |
| Constructors | Actions required to initialize [instances](#_Trm00045) of the class or the class itself |
| Destructors | Actions to perform before [instances](#_Trm00045) of the class are permanently discarded |
| Types | Nested [types](#_Trm00011) 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](#_Trm00012) 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](#_Trm00027) 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](#_Trm00012) of its base class. Inheritance means that a class implicitly contains all [members](#_Trm00012) 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](#_Trm00012) 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](#_Trm00024). 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](#_Trm00053) identifies exactly one storage location. No matter how many [instances](#_Trm00045) of a class are created, there is only ever one copy of a [static field](#_Trm00053).

A field declared without the static modifier defines an ***instance field***. Every instance of a class contains a separate copy of all the [instance field](#_Trm00054)s of that class.

In the following example, each instance of the Color class has a separate copy of the r, g, and b [instance field](#_Trm00054)s, but there is only one copy of the Black, White, Red, Green, and Blue [static field](#_Trm00053)s:

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](#_Trm00045) of the class.

Methods have a (possibly empty) list of ***parameters***, which represent values or variable references passed to the [method](#_Trm00056), and a ***return type***, which specifies the type of the value computed and returned by the [method](#_Trm00056). A [method](#_Trm00056)'s [return type](#_Trm00060) is void if it does not return a value.

Like [types](#_Trm00011), [method](#_Trm00056)s may also have a set of type [parameters](#_Trm00059), for which type arguments must be specified when the [method](#_Trm00056) is called. Unlike [types](#_Trm00011), the type arguments can often be inferred from the arguments of a [method](#_Trm00056) call and need not be explicitly given.

The ***signature*** of a [method](#_Trm00056) must be unique in the class in which the [method](#_Trm00056) is declared. The [signature](#_Trm00061) of a [method](#_Trm00056) consists of the name of the [method](#_Trm00056), the number of type [parameters](#_Trm00059) and the number, modifiers, and [types](#_Trm00011) of its [parameters](#_Trm00059). The [signature](#_Trm00061) of a [method](#_Trm00056) does not include the [return type](#_Trm00060).

#### Parameters

Parameters are used to pass values or variable references to [method](#_Trm00056)s. The [parameters](#_Trm00059) of a [method](#_Trm00056) get their actual values from the ***arguments*** that are specified when the [method](#_Trm00056) is invoked. There are four kinds of [parameters](#_Trm00059): value [parameters](#_Trm00059), reference [parameters](#_Trm00059), output [parameters](#_Trm00059), and parameter arrays.

A ***value parameter*** is used for input parameter passing. A [value parameter](#_Trm00063) corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a [value parameter](#_Trm00063) do not affect the argument that was passed for the parameter.

Value [parameters](#_Trm00059) can be optional, by specifying a default value so that corresponding [arguments](#_Trm00062) can be omitted.

A ***reference parameter*** is used for both input and output parameter passing. The argument passed for a [reference parameter](#_Trm00064) must be a variable, and during execution of the [method](#_Trm00056), the [reference parameter](#_Trm00064) represents the same storage location as the argument variable. A [reference parameter](#_Trm00064) is declared with the ref modifier. The following example shows the use of ref [parameters](#_Trm00059).

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](#_Trm00065) passing. An [output parameter](#_Trm00065) is similar to a [reference parameter](#_Trm00064) except that the initial value of the caller-provided argument is unimportant. An [output parameter](#_Trm00065) is declared with the out modifier. The following example shows the use of out [parameters](#_Trm00059).

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](#_Trm00062) to be passed to a [method](#_Trm00056). A [parameter array](#_Trm00066) is declared with the params modifier. Only the last parameter of a [method](#_Trm00056) can be a [parameter array](#_Trm00066), and the type of a [parameter array](#_Trm00066) must be a single-dimensional array type. The Write and WriteLine [method](#_Trm00056)s of the System.Console class are good examples of [parameter array](#_Trm00066) 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](#_Trm00056) that uses a [parameter array](#_Trm00066), the [parameter array](#_Trm00066) behaves exactly like a regular parameter of an array type. However, in an invocation of a [method](#_Trm00056) with a [parameter array](#_Trm00066), it is possible to pass either a single argument of the [parameter array](#_Trm00066) type or any number of [arguments](#_Trm00062) of the element type of the [parameter array](#_Trm00066). In the latter case, an array instance is automatically created and initialized with the given [arguments](#_Trm00062). 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](#_Trm00031)

A [method](#_Trm00056)'s body specifies the [statements](#_Trm00037) to execute when the [method](#_Trm00056) is invoked.

A [method](#_Trm00056) body can declare [variables](#_Trm00031) that are specific to the invocation of the [method](#_Trm00056). Such [variables](#_Trm00031) 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](#_Trm00068) at those points in the program.

A [method](#_Trm00056) can use return [statements](#_Trm00037) to return control to its caller. In a [method](#_Trm00056) returning void, return [statements](#_Trm00037) cannot specify an expression. In a [method](#_Trm00056) returning non-void, return [statements](#_Trm00037) must include an expression that computes the return value.

#### Static and instance [method](#_Trm00056)s

A [method](#_Trm00056) declared with a static modifier is a ***static method***. A static [method](#_Trm00056) does not operate on a specific instance and can only directly access static [members](#_Trm00012).

A [method](#_Trm00056) declared without a static modifier is an ***instance method***. An [instance method](#_Trm00070) operates on a specific instance and can access both static and [instance members](#_Trm00052). The instance on which an [instance method](#_Trm00070) was invoked can be explicitly accessed as this. It is an error to refer to this in a static [method](#_Trm00056).

The following Entity class has both static and [instance members](#_Trm00052).

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](#_Trm00070)) 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](#_Trm00054) and the nextSerialNo [static field](#_Trm00053).

The GetNextSerialNo and SetNextSerialNo static [method](#_Trm00056)s can access the nextSerialNo [static field](#_Trm00053), but it would be an error for them to directly access the serialNo [instance field](#_Trm00054).

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 [method](#_Trm00056)s are invoked on the class whereas the GetSerialNo [instance method](#_Trm00070) is invoked on [instances](#_Trm00045) of the class.

#### Virtual, override, and abstract [method](#_Trm00056)s

When an [instance method](#_Trm00070) declaration includes a virtual modifier, the [method](#_Trm00056) is said to be a ***virtual method***. When no virtual modifier is present, the [method](#_Trm00056) is said to be a ***non-virtual method***.

When a virtual [method](#_Trm00056) is invoked, the ***run-time type*** of the instance for which that invocation takes place determines the actual [method](#_Trm00056) implementation to invoke. In a nonvirtual [method](#_Trm00056) invocation, the ***compile-time type*** of the instance is the determining factor.

A virtual [method](#_Trm00056) can be ***overridden*** in a derived class. When an [instance method](#_Trm00070) declaration includes an override modifier, the [method](#_Trm00056) overrides an inherited virtual [method](#_Trm00056) with the same [signature](#_Trm00061). Whereas a virtual [method](#_Trm00056) declaration introduces a new [method](#_Trm00056), an override [method](#_Trm00056) declaration specializes an existing inherited virtual [method](#_Trm00056) by providing a new implementation of that [method](#_Trm00056).

An ***abstract*** [method](#_Trm00056) is a virtual [method](#_Trm00056) with no implementation. An [abstract](#_Trm00076) [method](#_Trm00056) is declared with the abstract modifier and is permitted only in a class that is also declared abstract. An [abstract](#_Trm00076) [method](#_Trm00056) must be [overridden](#_Trm00075) in every non-[abstract](#_Trm00076) derived class.

The following example declares an [abstract](#_Trm00076) class, Expression, which represents an expression tree node, and three [derived classes](#_Trm00049), 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](#_Trm00011) introduced in [§4.6](#_Toc00119)).

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](#_Trm00045) 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](#_Trm00056) of an Expression instance is invoked to evaluate the given expression and produce a double value. The [method](#_Trm00056) 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](#_Trm00056) is a virtual [abstract](#_Trm00076) [method](#_Trm00056), meaning that non-[abstract](#_Trm00076) [derived classes](#_Trm00049) 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](#_Trm00033) (by recursively invoking their Evaluate [method](#_Trm00056)s) 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 [method](#_Trm00056)s in the same class to have the same name as long as they have unique [signature](#_Trm00061)s. When compiling an invocation of an [overloaded](#_Trm00036) [method](#_Trm00056), the compiler uses ***overload resolution*** to determine the specific [method](#_Trm00056) to invoke. Overload resolution finds the one [method](#_Trm00056) that best matches the [arguments](#_Trm00062) or reports an error if no single best match can be found. The following example shows [overload resolution](#_Trm00078) in effect. The comment for each invocation in the Main [method](#_Trm00056) shows which [method](#_Trm00056) 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](#_Trm00056) can always be selected by explicitly casting the [arguments](#_Trm00062) to the exact parameter [types](#_Trm00011) and/or explicitly supplying type [arguments](#_Trm00062).

### Other function [members](#_Trm00012)

Members that contain executable code are collectively known as the ***function members*** of a class. The preceding section describes [method](#_Trm00056)s, which are the primary kind of [function members](#_Trm00079). This section describes the other kinds of [function members](#_Trm00079) supported by C#: constructors, properties, indexers, events, [operators](#_Trm00034), and destructors.

The following code shows a generic class called List<T>, which implements a growable list of [objects](#_Trm00046). The class contains several examples of the most common kinds of [function members](#_Trm00079).

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](#_Trm00056) with no [return type](#_Trm00060) and the same name as the containing class. If a constructor declaration includes a static modifier, it declares a [static constructor](#_Trm00081). Otherwise, it declares an [instance constructor](#_Trm00080).

Instance constructors can be [overloaded](#_Trm00036). For example, the List<T> class declares two [instance constructor](#_Trm00080)s, one with no [parameters](#_Trm00059) and one that takes an int parameter. Instance constructors are invoked using the new operator. The following [statements](#_Trm00037) allocate two List<string> [instances](#_Trm00045) 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](#_Trm00012), [instance constructor](#_Trm00080)s are not inherited, and a class has no [instance constructor](#_Trm00080)s other than those actually declared in the class. If no [instance constructor](#_Trm00080) is supplied for a class, then an empty one with no [parameters](#_Trm00059) is automatically provided.

#### Properties

***Properties*** are a natural extension of fields. Both are named [members](#_Trm00012) with associated [types](#_Trm00011), 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](#_Trm00037) 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](#_Trm00056) 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](#_Trm00056) with a single parameter named value and no [return type](#_Trm00060). 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 [method](#_Trm00056)s, 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](#_Trm00046) to be indexed in the same way as an array. An [indexer](#_Trm00087) 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](#_Trm00059) are available in the accessor(s) of the [indexer](#_Trm00087). Similar to properties, [indexer](#_Trm00087)s can be read-write, read-only, and write-only, and the accessor(s) of an [indexer](#_Trm00087) can be virtual.

The List class declares a single read-write [indexer](#_Trm00087) that takes an int parameter. The [indexer](#_Trm00087) makes it possible to index List [instances](#_Trm00045) 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](#_Trm00036), meaning that a class can declare multiple [indexer](#_Trm00087)s as long as the number or [types](#_Trm00011) of their [parameters](#_Trm00059) differ.

#### Events

An ***event*** is a member that enables a class or object to provide notifications. An [event](#_Trm00088) 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](#_Trm00088) member, the [event](#_Trm00088) behaves just like a field of a delegate type (provided the [event](#_Trm00088) is not [abstract](#_Trm00076) and does not declare [accessors](#_Trm00083)). The field stores a reference to a delegate that represents the [event](#_Trm00088) handlers that have been added to the [event](#_Trm00088). If no [event](#_Trm00088) handles are present, the field is null.

The List<T> class declares a single [event](#_Trm00088) member called Changed, which indicates that a new item has been added to the list. The Changed [event](#_Trm00088) is raised by the OnChanged virtual [method](#_Trm00056), which first checks whether the [event](#_Trm00088) is null (meaning that no handlers are present). The notion of raising an [event](#_Trm00088) is precisely equivalent to invoking the delegate represented by the [event](#_Trm00088)—thus, there are no special language constructs for raising [event](#_Trm00088)s.

Clients react to [event](#_Trm00088)s through ***event handlers***. Event handlers are attached using the += operator and removed using the -= operator. The following example attaches an [event](#_Trm00088) handler to the Changed [event](#_Trm00088) 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](#_Trm00088) is desired, an [event](#_Trm00088) declaration can explicitly provide add and remove [accessors](#_Trm00083), 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](#_Trm00090) to [instances](#_Trm00045) of a class. Three kinds of [operator](#_Trm00090)s can be defined: unary [operator](#_Trm00090)s, binary [operator](#_Trm00090)s, and conversion [operator](#_Trm00090)s. All [operator](#_Trm00090)s must be declared as public and static.

The List<T> class declares two [operator](#_Trm00090)s, operator== and operator!=, and thus gives new meaning to expressions that apply those [operator](#_Trm00090)s to List [instances](#_Trm00045). Specifically, the [operator](#_Trm00090)s define equality of two List<T> [instances](#_Trm00045) as comparing each of the contained [objects](#_Trm00046) using their Equals [method](#_Trm00056)s. The following example uses the == [operator](#_Trm00090) to compare two List<int> [instances](#_Trm00045).

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](#_Trm00046) 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](#_Trm00045).

#### Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. Destructors cannot have [parameters](#_Trm00059), they cannot have accessibility modifiers, and they cannot be invoked explicitly. The [destructor](#_Trm00091) for an instance is invoked automatically during garbage collection.

The garbage collector is allowed wide latitude in deciding when to collect [objects](#_Trm00046) and run [destructor](#_Trm00091)s. Specifically, the timing of [destructor](#_Trm00091) invocations is not deterministic, and [destructor](#_Trm00091)s may be executed on any thread. For these and other reasons, classes should implement [destructor](#_Trm00091)s 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](#_Trm00012) and [function members](#_Trm00079), but unlike classes, [structs](#_Trm00092) are value [types](#_Trm00011) 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](#_Trm00011) do not support user-specified [inheritance](#_Trm00047), and all [struct types](#_Trm00022) 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](#_Trm00092). The use of [structs](#_Trm00092) 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](#_Trm00046) 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](#_Trm00045) are stored in-line in the array.

Struct constructors are invoked with the new [operator](#_Trm00090), 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](#_Trm00031) to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With [structs](#_Trm00092), the [variables](#_Trm00031) 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](#_Trm00092). First, copying an entire struct is typically less efficient than copying an object reference, so assignment and [value parameter](#_Trm00063) passing can be more expensive with [structs](#_Trm00092) than with [reference types](#_Trm00019). Second, except for ref and out [parameters](#_Trm00059), it is not possible to create references to [structs](#_Trm00092), which rules out their usage in a number of situations.

## Arrays

An ***array*** is a data structure that contains a number of [variables](#_Trm00031) that are accessed through computed indices. The [variables](#_Trm00031) contained in an [array](#_Trm00093), also called the ***elements*** of the [array](#_Trm00093), are all of the same type, and this type is called the ***element type*** of the [array](#_Trm00093).

Array [types](#_Trm00011) are [reference types](#_Trm00019), and the declaration of an [array](#_Trm00093) variable simply sets aside space for a reference to an [array](#_Trm00093) instance. Actual [array](#_Trm00093) [instances](#_Trm00045) are created dynamically at run-time using the new [operator](#_Trm00090). The new operation specifies the ***length*** of the new [array](#_Trm00093) instance, which is then fixed for the lifetime of the instance. The indices of the [elements](#_Trm00094) of an [array](#_Trm00093) range from 0 to Length - 1. The new [operator](#_Trm00090) automatically initializes the [elements](#_Trm00094) of an [array](#_Trm00093) to their default value, which, for example, is zero for all numeric [types](#_Trm00011) and null for all [reference types](#_Trm00019).

The following example creates an [array](#_Trm00093) of int [elements](#_Trm00094), initializes the [array](#_Trm00093), and prints out the contents of the [array](#_Trm00093).

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](#_Trm00093) type, also known as the ***rank*** of the [array](#_Trm00093) type, is one plus the number of commas written between the square brackets of the [array](#_Trm00093) type. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional [array](#_Trm00093).

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

The a1 [array](#_Trm00093) contains 10 [elements](#_Trm00094), the a2 [array](#_Trm00093) contains 50 (10 × 5) [elements](#_Trm00094), and the a3 [array](#_Trm00093) contains 100 (10 × 5 × 2) [elements](#_Trm00094).

The [element type](#_Trm00095) of an [array](#_Trm00093) can be any type, including an [array](#_Trm00093) type. An [array](#_Trm00093) with [elements](#_Trm00094) of an [array](#_Trm00093) type is sometimes called a ***jagged array*** because the [length](#_Trm00096)s of the element [array](#_Trm00093)s do not all have to be the same. The following example allocates an [array](#_Trm00093) of [array](#_Trm00093)s 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](#_Trm00093) with three [elements](#_Trm00094), each of type int[] and each with an initial value of null. The subsequent lines then initialize the three [elements](#_Trm00094) with references to individual [array](#_Trm00093) [instances](#_Trm00045) of varying [length](#_Trm00096)s.

The new [operator](#_Trm00090) permits the initial values of the [array](#_Trm00093) [elements](#_Trm00094) 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](#_Trm00094).

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

Note that the [length](#_Trm00096) of the [array](#_Trm00093) is inferred from the number of expressions between { and }. Local variable and field declarations can be shortened further such that the [array](#_Trm00093) 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](#_Trm00092). An [interface](#_Trm00102) can contain [method](#_Trm00056)s, properties, [event](#_Trm00088)s, and [indexer](#_Trm00087)s. An [interface](#_Trm00102) does not provide implementations of the [members](#_Trm00012) it defines—it merely specifies the [members](#_Trm00012) that must be supplied by classes or [structs](#_Trm00092) that implement the [interface](#_Trm00102).

Interfaces may employ ***multiple inheritance***. In the following example, the [interface](#_Trm00102) 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](#_Trm00044) and [structs](#_Trm00092) can implement multiple [interface](#_Trm00102)s. 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](#_Trm00102), [instances](#_Trm00045) of that class or struct can be implicitly converted to that [interface](#_Trm00102) 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](#_Trm00102), dynamic type casts can be used. For example, the following [statements](#_Trm00037) use dynamic type casts to obtain an object's IControl and IDataBound [interface](#_Trm00102) 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](#_Trm00056) from the IControl [interface](#_Trm00102) and the Bind [method](#_Trm00056) from the IDataBound [interface](#_Trm00102) are implemented using public [members](#_Trm00012). C# also supports ***explicit interface member implementations***, using which the class or struct can avoid making the [members](#_Trm00012) public. An explicit [interface](#_Trm00102) member implementation is written using the fully qualified [interface](#_Trm00102) member name. For example, the EditBox class could implement the IControl.Paint and IDataBound.Bind [method](#_Trm00056)s using [explicit interface member implementations](#_Trm00104) as follows.

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

Explicit [interface](#_Trm00102) [members](#_Trm00012) can only be accessed via the [interface](#_Trm00102) 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](#_Trm00102) 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](#_Trm00105) 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](#_Trm00105) has a corresponding integral type called the ***underlying type*** of the [enum type](#_Trm00105). An [enum type](#_Trm00105) that does not explicitly declare an [underlying type](#_Trm00106) has an [underlying type](#_Trm00106) of int. An [enum type](#_Trm00105)'s storage format and range of possible values are determined by its [underlying type](#_Trm00106). The set of values that an [enum type](#_Trm00105) can take on is not limited by its enum [members](#_Trm00012). In particular, any value of the [underlying type](#_Trm00106) of an enum can be cast to the [enum type](#_Trm00105) and is a distinct valid value of that [enum type](#_Trm00105).

The following example declares an [enum type](#_Trm00105) named Alignment with an [underlying type](#_Trm00106) 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](#_Trm00106) 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](#_Trm00105)) 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](#_Trm00105) is the integral value zero converted to the [enum type](#_Trm00105). In cases where [variables](#_Trm00031) are automatically initialized to a default value, this is the value given to [variables](#_Trm00031) of [enum type](#_Trm00105)s. In order for the default value of an [enum type](#_Trm00105) to be easily available, the literal 0 implicitly converts to any [enum type](#_Trm00105). Thus, the following is permitted.

Color c = 0;

## Delegates

A ***delegate type*** represents references to [method](#_Trm00056)s with a particular parameter list and [return type](#_Trm00060). Delegates make it possible to treat [method](#_Trm00056)s as entities that can be assigned to [variables](#_Trm00031) and passed as [parameters](#_Trm00059). 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](#_Trm00006).

The following example declares and uses a [delegate type](#_Trm00107) 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](#_Trm00107) can reference any [method](#_Trm00056) that takes a double argument and returns a double value. The Apply [method](#_Trm00056) applies a given Function to the [elements](#_Trm00094) of a double[], returning a double[] with the results. In the Main [method](#_Trm00056), Apply is used to apply three different functions to a double[].

A delegate can reference either a static [method](#_Trm00056) (such as Square or Math.Sin in the previous example) or an [instance method](#_Trm00070) (such as m.Multiply in the previous example). A delegate that references an [instance method](#_Trm00070) also references a particular object, and when the [instance method](#_Trm00070) is invoked through the delegate, that object becomes this in the invocation.

Delegates can also be created using anonymous functions, which are "inline [method](#_Trm00056)s" that are created on the fly. Anonymous functions can see the [local variables](#_Trm00067) of the sourrounding [method](#_Trm00056)s. 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](#_Trm00056) it references; all that matters is that the referenced [method](#_Trm00056) has the same [parameters](#_Trm00059) and [return type](#_Trm00060) as the delegate.

## Attributes

Types, [members](#_Trm00012), and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a [method](#_Trm00056) is controlled using the public, protected, internal, and private modifiers. C# generalizes this capability such that user-defined [types](#_Trm00011) 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](#_Trm00062), 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](#_Trm00056) 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

## Programs

A C# ***program*** consists of one or more ***source files***, known formally as ***compilation units*** ([§9.1](#_Toc00378)). 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](#_Trm00109) 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# [program](#_Trm00109)ming language using two grammars. The ***lexical grammar*** ([§2.2.2](#_Toc00036)) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The ***syntactic grammar*** ([§2.2.3](#_Toc00037)) defines how the tokens resulting from the [lexical grammar](#_Trm00112) are combined to form C# [program](#_Trm00109)s.

### Grammar notation

The lexical and [syntactic grammar](#_Trm00113)s are presented in Backus-Naur form using the notation of the ANTLR grammar tool.

### Lexical grammar

The [lexical grammar](#_Trm00112) of C# is presented in [§2.3](#_Toc00038), [§2.4](#_Toc00042), and [§2.5](#_Toc00054). The terminal symbols of the [lexical grammar](#_Trm00112) are the characters of the Unicode character set, and the [lexical grammar](#_Trm00112) specifies how characters are combined to form tokens ([§2.4](#_Toc00042)), white space ([§2.3.3](#_Toc00041)), comments ([§2.3.2](#_Toc00040)), and pre-processing directives ([§2.5](#_Toc00054)).

Every source file in a C# [program](#_Trm00109) must conform to the [input](#_Grm00001) production of the [lexical grammar](#_Trm00112) ([§2.3](#_Toc00038)).

### Syntactic grammar

The [syntactic grammar](#_Trm00113) of C# is presented in the chapters and appendices that follow this chapter. The terminal symbols of the [syntactic grammar](#_Trm00113) are the tokens defined by the [lexical grammar](#_Trm00112), and the [syntactic grammar](#_Trm00113) specifies how tokens are combined to form C# [program](#_Trm00109)s.

Every source file in a C# [program](#_Trm00109) must conform to the [compilation\_unit](#_Grm00098) production of the [syntactic grammar](#_Trm00113) ([§9.1](#_Toc00378)).

## Lexical analysis

The [input](#_Grm00001) production defines the lexical structure of a C# source file. Each source file in a C# [program](#_Trm00109) must conform to this [lexical grammar](#_Trm00112) 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](#_Trm00094) make up the lexical structure of a C# source file: Line terminators ([§2.3.1](#_Toc00039)), white space ([§2.3.3](#_Toc00041)), comments ([§2.3.2](#_Toc00040)), tokens ([§2.4](#_Toc00042)), and pre-processing directives ([§2.5](#_Toc00054)). Of these basic [elements](#_Trm00094), only tokens are significant in the [syntactic grammar](#_Trm00113) of a C# [program](#_Trm00109) ([§2.2.3](#_Toc00037)).

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](#_Trm00094) have no impact on the syntactic structure of a C# [program](#_Trm00109).

When several [lexical grammar](#_Trm00112) 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](#_Trm00109):

* 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](#_Trm00115) 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, [operator](#_Trm00090)s, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:  
 | identifier  
 | keyword  
 | integer\_literal  
 | real\_literal  
 | character\_literal  
 | string\_literal  
 | operator\_or\_punctuator  
 ;

### Unicode character escape sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers ([§2.4.2](#_Toc00044)), character literals ([§2.4.4.4](#_Toc00050)), and regular string literals ([§2.4.4.5](#_Toc00051)). A Unicode character escape is not processed in any other location (for example, to form an [operator](#_Trm00090), 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](#_Trm00109) 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 [program](#_Trm00109)ming 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](#_Trm00109) 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 [program](#_Trm00109)ming 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](#_Trm00056) 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](#_Grm00006) is transformed into its corresponding Unicode character.
* Any [formatting\_character](#_Grm00007)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 [keyword](#_Trm00117)s. Such identifiers are sometimes referred to as "contextual [keyword](#_Trm00117)s". For example, within a property declaration, the "get" and "set" identifiers have special meaning ([§10.7.2](#_Toc00459)). 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 ([§8.5.1](#_Toc00355)), a contectual [keyword](#_Trm00117) can conflict with declared names. In such cases, the declared name takes [precedence](#_Trm00035) over the use of the identifier as a contextual [keyword](#_Trm00117).

### 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 [literal](#_Trm00118)s

There are two boolean [literal](#_Trm00118) values: true and false.

boolean\_literal:  
 | 'true'  
 | 'false'  
 ;

The type of a [boolean\_literal](#_Grm00010) is bool.

#### Integer [literal](#_Trm00118)s

Integer [literal](#_Trm00118)s are used to write values of [types](#_Trm00011) int, uint, long, and ulong. Integer [literal](#_Trm00118)s 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](#_Trm00118) is determined as follows:

* If the [literal](#_Trm00118) has no suffix, it has the first of these [types](#_Trm00011) in which its value can be represented: int, uint, long, ulong.
* If the [literal](#_Trm00118) is suffixed by U or u, it has the first of these [types](#_Trm00011) in which its value can be represented: uint, ulong.
* If the [literal](#_Trm00118) is suffixed by L or l, it has the first of these [types](#_Trm00011) in which its value can be represented: long, ulong.
* If the [literal](#_Trm00118) 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](#_Trm00118) 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 [literal](#_Trm00118)s 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 [literal](#_Trm00118)s, the following two rules exist:

* When a [decimal\_integer\_literal](#_Grm00011) with the value 2147483648 (2^31) and no [integer\_type\_suffix](#_Grm00011) appears as the token immediately following a unary minus [operator](#_Trm00090) token ([§7.7.2](#_Toc00283)), the result is a constant of type int with the value -2147483648 (-2^31). In all other situations, such a [decimal\_integer\_literal](#_Grm00011) is of type uint.
* When a [decimal\_integer\_literal](#_Grm00011) with the value 9223372036854775808 (2^63) and no [integer\_type\_suffix](#_Grm00011) or the [integer\_type\_suffix](#_Grm00011) L or l appears as the token immediately following a unary minus [operator](#_Trm00090) token ([§7.7.2](#_Toc00283)), the result is a constant of type long with the value -9223372036854775808 (-2^63). In all other situations, such a [decimal\_integer\_literal](#_Grm00011) is of type ulong.

#### Real [literal](#_Trm00118)s

Real [literal](#_Trm00118)s are used to write values of [types](#_Trm00011) 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](#_Grm00012) is specified, the type of the real [literal](#_Trm00118) is double. Otherwise, the real type suffix determines the type of the real [literal](#_Trm00118), as follows:

* A real [literal](#_Trm00118) suffixed by F or f is of type float. For example, the [literal](#_Trm00118)s 1f, 1.5f, 1e10f, and 123.456F are all of type float.
* A real [literal](#_Trm00118) suffixed by D or d is of type double. For example, the [literal](#_Trm00118)s 1d, 1.5d, 1e10d, and 123.456D are all of type double.
* A real [literal](#_Trm00118) suffixed by M or m is of type decimal. For example, the [literal](#_Trm00118)s 1m, 1.5m, 1e10m, and 123.456M are all of type decimal. This [literal](#_Trm00118) is converted to a decimal value by taking the exact value, and, if necessary, rounding to the nearest representable value using banker's rounding ([§4.1.7](#_Toc00098)). Any scale apparent in the [literal](#_Trm00118) 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](#_Trm00118) 2.900m will be parsed to form the decimal with sign 0, coefficient 2900, and scale 3.

If the specified [literal](#_Trm00118) cannot be represented in the indicated type, a compile-time error occurs.

The value of a real [literal](#_Trm00118) of type float or double is determined by using the IEEE "round to nearest" mode.

Note that in a real [literal](#_Trm00118), decimal digits are always required after the decimal point. For example, 1.3F is a real [literal](#_Trm00118) but 1.F is not.

#### Character [literal](#_Trm00118)s

A character [literal](#_Trm00118) 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](#_Trm00118) 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](#_Grm00013) 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](#_Trm00118) is greater than U+FFFF, a compile-time error occurs.

A Unicode character escape sequence ([§2.4.1](#_Toc00043)) in a character [literal](#_Trm00118) must be in the range U+0000 to U+FFFF.

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

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

The type of a [character\_literal](#_Grm00013) is char.

#### String [literal](#_Trm00118)s

C# supports two forms of string [literal](#_Trm00118)s: ***regular string literals*** and ***verbatim string literals***.

A regular string [literal](#_Trm00118) 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](#_Trm00118) 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](#_Trm00118), the characters between the delimiters are interpreted verbatim, the only exception being a [quote\_escape\_sequence](#_Grm00014). In particular, simple escape sequences, and hexadecimal and Unicode escape sequences are not processed in verbatim string [literal](#_Trm00118)s. A verbatim string [literal](#_Trm00118) 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](#_Grm00014) 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 [literal](#_Trm00118)s. The last string [literal](#_Trm00118), j, is a verbatim string [literal](#_Trm00118) 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](#_Trm00118) "\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](#_Grm00014) is string.

Each string [literal](#_Trm00118) does not necessarily result in a new string instance. When two or more string [literal](#_Trm00118)s that are equivalent according to the string equality [operator](#_Trm00090) ([§7.10.7](#_Toc00306)) appear in the same [program](#_Trm00109), these string [literal](#_Trm00118)s 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 [literal](#_Trm00118)s refer to the same string instance.

#### The null [literal](#_Trm00118)

null\_literal:  
 | 'null'  
 ;

The [null\_literal](#_Grm00015) can be implicitly converted to a reference type or nullable type.

### Operators and punctuators

There are several kinds of [operator](#_Trm00090)s and punctuators. Operators are used in expressions to describe operations involving one or more [operands](#_Trm00033). For example, the expression a + b uses the + [operator](#_Trm00090) to add the two [operands](#_Trm00033) a and b. Punctuators are for grouping and separating.

operator\_or\_punctuator:  
 | '{' | '}' | '[' | ']' | '(' | ')' | '.' | ',' | ':' | ';'  
 | '+' | '-' | '\*' | '/' | '%' | '&' | '|' | '^' | '!' | '~'  
 | '=' | '<' | '>' | '?' | '??' | '::' | '++' | '--' | '&&' | '||'  
 | '->' | '==' | '!=' | '<=' | '>=' | '+=' | '-=' | '\*=' | '/=' | '%='  
 | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'  
 ;  
  
right\_shift:  
 | '>>'  
 ;  
  
right\_shift\_assignment:  
 | '>>='  
 ;

The vertical bar in the [right\_shift](#_Grm00016) and [right\_shift\_assignment](#_Grm00016) productions are used to indicate that, unlike other productions in the [syntactic grammar](#_Trm00113), 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\_list](#_Grm00109)s ([§10.1.3](#_Toc00395)).

## Pre-processing directives

The pre-processing directives provide the ability to conditionally skip sections of [source files](#_Trm00110), 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++ [program](#_Trm00109)ming 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 ([§2.5.3](#_Toc00057)).
* #if, #elif, #else, and #endif, which are used to conditionally skip sections of source code ([§2.5.4](#_Toc00058)).
* #line, which is used to control line numbers emitted for errors and warnings ([§2.5.7](#_Toc00061)).
* #error and #warning, which are used to issue errors and warnings, respectively ([§2.5.5](#_Toc00059)).
* #region and #endregion, which are used to explicitly mark sections of source code ([§2.5.6](#_Toc00060)).
* #pragma, which is used to specify optional contextual information to the compiler ([§2.5.8](#_Toc00062)).

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](#_Trm00115) (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](#_Trm00113) 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](#_Trm00109). For example, when compiled, the [program](#_Trm00109):

#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](#_Trm00109):

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

Thus, whereas lexically, the two [program](#_Trm00109)s 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 ([§2.5.2](#_Toc00056)) 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 un[defined](#_Trm00121) unless it has been explicitly [defined](#_Trm00121) 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](#_Trm00121) in that source file. The symbol remains [defined](#_Trm00121) 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](#_Trm00110) in the same [program](#_Trm00109).

When referenced in a pre-processing expression, a [defined](#_Trm00121) conditional compilation symbol has the boolean value true, and an un[defined](#_Trm00121) 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 un[defined](#_Trm00121) 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](#_Trm00109). 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 [operator](#_Trm00090)s !, ==, !=, && 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](#_Trm00121) conditional compilation symbol has the boolean value true, and an un[defined](#_Trm00121) 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 ([§7.19](#_Toc00346)), except that the only user-[defined](#_Trm00121) 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](#_Trm00121), starting with the source line that follows the directive. Likewise, the processing of an #undef directive causes the given conditional compilation symbol to become un[defined](#_Trm00121), starting with the source line that follows the directive.

Any #define and #undef directives in a source file must occur before the first [token](#_Grm00005) ([§2.4](#_Toc00042)) 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](#_Trm00117)) 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](#_Trm00121), 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](#_Trm00121). 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](#_Grm00021) selects at most one of the contained [conditional\_section](#_Grm00021)s for normal lexical processing:

* The [pp\_expression](#_Grm00019)s of the #if and #elif directives are evaluated in order until one yields true. If an expression yields true, the [conditional\_section](#_Grm00021) of the corresponding directive is selected.
* If all [pp\_expression](#_Grm00019)s yield false, and if an #else directive is present, the [conditional\_section](#_Grm00021) of the #else directive is selected.
* Otherwise, no [conditional\_section](#_Grm00021) is selected.

The selected [conditional\_section](#_Grm00021), if any, is processed as a normal [input\_section](#_Grm00001): the source code contained in the section must adhere to the [lexical grammar](#_Trm00112); 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](#_Grm00021)s, if any, are processed as [skipped\_section](#_Grm00021)s: except for pre-processing directives, the source code in the section need not adhere to the [lexical grammar](#_Trm00112); 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](#_Grm00021) that is being processed as a [skipped\_section](#_Grm00021), any nested [conditional\_section](#_Grm00021)s (contained in nested #if...#endif and #region...#endregion con[structs](#_Trm00092)) are also processed as [skipped\_section](#_Grm00021)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](#_Trm00094). For example, the [program](#_Trm00109):

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](#_Grm00019). The example:

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

always produces the same token stream (class Q { }), regardless of whether or not X is [defined](#_Trm00121). If X is [defined](#_Trm00121), the only processed directives are #if and #endif, due to the multi-line comment. If X is un[defined](#_Trm00121), 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](#_Trm00121). Note that a [pp\_message](#_Grm00022) 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 [program](#_Trm00109)mer 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](#_Grm00022)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](#_Trm00108) ([§17.4.4](#_Toc00582)).

Line directives are most commonly used in meta-[program](#_Trm00109)ming 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](#_Grm00024) 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](#_Grm00024) differs from a regular string [literal](#_Trm00118) in that escape characters are not processed; the "\" character simply designates an ordinary backslash character within a [file\_name](#_Grm00024).

### 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](#_Trm00109) 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](#_Trm00109) 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](#_Trm00012) 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

## Application Startup

An assembly that has an ***entry point*** is called an ***application***. When an [application](#_Trm00124) is run, a new ***application domain*** is created. Several different instantiations of an [application](#_Trm00124) may exist on the same machine at the same time, and each has its own [application](#_Trm00124) domain.

An [application](#_Trm00124) domain enables [application](#_Trm00124) isolation by acting as a container for [application](#_Trm00124) state. An [application](#_Trm00124) domain acts as a container and boundary for the [types](#_Trm00011) [defined](#_Trm00121) in the [application](#_Trm00124) and the class [libraries](#_Trm00015) it uses. Types loaded into one [application](#_Trm00124) domain are distinct from the same type loaded into another [application](#_Trm00124) domain, and [instances](#_Trm00045) of [objects](#_Trm00046) are not directly shared between [application](#_Trm00124) domains. For instance, each [application](#_Trm00124) domain has its own copy of static [variables](#_Trm00031) for these [types](#_Trm00011), and a [static constructor](#_Trm00081) for a type is run at most once per [application](#_Trm00124) domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of [application](#_Trm00124) domains.

***Application startup*** occurs when the execution environment calls a designated [method](#_Trm00056), which is referred to as the [application](#_Trm00124)'s [entry point](#_Trm00123). This [entry point](#_Trm00123) [method](#_Trm00056) is always named Main, and can have one of the following [signature](#_Trm00061)s:

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

As shown, the [entry point](#_Trm00123) may optionally return an int value. This return value is used in [application](#_Trm00124) termination ([§3.2](#_Toc00066)).

The [entry point](#_Trm00123) 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](#_Trm00062) that were specified when the [application](#_Trm00124) was started. The string[] argument is never null, but it may have a [length](#_Trm00096) of zero if no command-line [arguments](#_Trm00062) were specified.

Since C# supports [method](#_Trm00056) [overloading](#_Trm00077), a class or struct may contain multiple definitions of some [method](#_Trm00056), provided each has a different [signature](#_Trm00061). However, within a single [program](#_Trm00109), no class or struct may contain more than one [method](#_Trm00056) called Main whose definition qualifies it to be used as an [application](#_Trm00124) [entry point](#_Trm00123). Other [overloaded](#_Trm00036) versions of Main are permitted, however, provided they have more than one parameter, or their only parameter is other than type string[].

An [application](#_Trm00124) can be made up of multiple classes or [structs](#_Trm00092). It is possible for more than one of these classes or [structs](#_Trm00092) to contain a [method](#_Trm00056) called Main whose definition qualifies it to be used as an [application](#_Trm00124) [entry point](#_Trm00123). In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these Main [method](#_Trm00056)s as the [entry point](#_Trm00123).

In C#, every [method](#_Trm00056) must be [defined](#_Trm00121) as a member of a class or struct. Ordinarily, the declared accessibility ([§3.5.1](#_Toc00077)) of a [method](#_Trm00056) is determined by the access modifiers ([§10.3.5](#_Toc00415)) 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](#_Trm00056) of a given type to be callable, both the type and the member must be accessible. However, the [application](#_Trm00124) [entry point](#_Trm00123) is a special case. Specifically, the execution environment can access the [application](#_Trm00124)'s [entry point](#_Trm00123) regardless of its declared accessibility and regardless of the declared accessibility of its enclosing [type declarations](#_Trm00028).

The [application](#_Trm00124) [entry point](#_Trm00123) [method](#_Trm00056) may not be in a generic class declaration.

In all other respects, [entry point](#_Trm00123) [method](#_Trm00056)s behave like those that are not [entry point](#_Trm00123)s.

## Application termination

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

If the [return type](#_Trm00060) of the [application](#_Trm00124)'s ***entry point*** [method](#_Trm00056) is int, the value returned serves as the [application](#_Trm00124)'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](#_Trm00060) of the [entry point](#_Trm00123) [method](#_Trm00056) is void, reaching the right brace (}) which terminates that [method](#_Trm00056), or executing a return statement that has no expression, results in a [termination status code](#_Trm00129) of 0.

Prior to an [application](#_Trm00124)'s termination, [destructor](#_Trm00091)s for all of its [objects](#_Trm00046) that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library [method](#_Trm00056) GC.SuppressFinalize, for example).

## Declarations

Declarations in a C# [program](#_Trm00109) define the constituent [elements](#_Trm00094) of the [program](#_Trm00109). C# [program](#_Trm00109)s are organized using [namespaces](#_Trm00010) ([§9](#_Toc00377)), which can contain [type declarations](#_Trm00028) and nested namespace declarations. Type declarations ([§9.6](#_Toc00385)) are used to define classes ([§10](#_Toc00388)), [structs](#_Trm00092) ([§10.14](#_Toc00483)), [interface](#_Trm00102)s ([§13](#_Toc00527)), enums ([§14](#_Toc00551)), and delegates ([§15](#_Toc00557)). The kinds of [members](#_Trm00012) permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants ([§10.4](#_Toc00430)), fields ([§10.5](#_Toc00431)), [method](#_Trm00056)s ([§10.6](#_Toc00441)), properties ([§10.7](#_Toc00457)), [event](#_Trm00088)s ([§10.8](#_Toc00463)), [indexer](#_Trm00087)s ([§10.9](#_Toc00468)), [operator](#_Trm00090)s ([§10.10](#_Toc00470)), [instance constructor](#_Trm00080)s ([§10.11](#_Toc00474)), [static constructor](#_Trm00081)s ([§10.12](#_Toc00481)), [destructor](#_Trm00091)s ([§10.13](#_Toc00482)), and nested [types](#_Trm00011)([§10.3.8](#_Toc00418)).

A declaration defines a name in the ***declaration space*** to which the declaration belongs. Except for [overloaded](#_Trm00036) [members](#_Trm00012) ([§3.6](#_Toc00081)), it is a compile-time error to have two or more declarations that introduce [members](#_Trm00012) with the same name in a [declaration space](#_Trm00130). It is never possible for a [declaration space](#_Trm00130) to contain different kinds of [members](#_Trm00012) with the same name. For example, a [declaration space](#_Trm00130) can never contain a field and a [method](#_Trm00056) by the same name.

There are several different [types](#_Trm00011) of [declaration space](#_Trm00130)s, as described in the following.

* Within all [source files](#_Trm00110) of a [program](#_Trm00109), [namespace\_member\_declaration](#_Grm00104)s with no enclosing [namespace\_declaration](#_Grm00099) are [members](#_Trm00012) of a single combined [declaration space](#_Trm00130) called the ***global declaration space***.
* Within all [source files](#_Trm00110) of a [program](#_Trm00109), [namespace\_member\_declaration](#_Grm00104)s within [namespace\_declaration](#_Grm00099)s that have the same fully qualified namespace name are [members](#_Trm00012) of a single combined [declaration space](#_Trm00130).
* Each class, struct, or [interface](#_Trm00102) declaration creates a new [declaration space](#_Trm00130). Names are introduced into this [declaration space](#_Trm00130) through [class\_member\_declaration](#_Grm00113)s, [struct\_member\_declaration](#_Grm00130)s, [interface\_member\_declaration](#_Grm00138)s, or [type\_parameter](#_Grm00032)s. Except for [overloaded](#_Trm00036) [instance constructor](#_Trm00080) declarations and [static constructor](#_Trm00081) declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or [interface](#_Trm00102) permits the declaration of [overloaded](#_Trm00036) [method](#_Trm00056)s and [indexer](#_Trm00087)s. Furthermore, a class or struct permits the declaration of [overloaded](#_Trm00036) [instance constructor](#_Trm00080)s and [operator](#_Trm00090)s. For example, a class, struct, or [interface](#_Trm00102) may contain multiple [method](#_Trm00056) declarations with the same name, provided these [method](#_Trm00056) declarations differ in their [signature](#_Trm00061) ([§3.6](#_Toc00081)). Note that [base classes](#_Trm00050) do not contribute to the [declaration space](#_Trm00130) of a class, and base [interface](#_Trm00102)s do not contribute to the [declaration space](#_Trm00130) of an [interface](#_Trm00102). Thus, a derived class or [interface](#_Trm00102) 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](#_Trm00130). Names are introduced into this [declaration space](#_Trm00130) through formal [parameters](#_Trm00059) ([fixed\_parameter](#_Grm00117)s and [parameter\_array](#_Grm00117)s) and [type\_parameter](#_Grm00032)s.
* Each enumeration declaration creates a new [declaration space](#_Trm00130). Names are introduced into this [declaration space](#_Trm00130) through [enum\_member\_declarations](#_Grm00145).
* Each [method](#_Trm00056) declaration, [indexer](#_Trm00087) declaration, [operator](#_Trm00090) declaration, [instance constructor](#_Trm00080) declaration and anonymous function creates a new [declaration space](#_Trm00130) called a ***local variable declaration space***. Names are introduced into this [declaration space](#_Trm00130) through formal [parameters](#_Trm00059) ([fixed\_parameter](#_Grm00117)s and [parameter\_array](#_Grm00117)s) and [type\_parameter](#_Grm00032)s. The body of the function member or anonymous function, if any, is considered to be nested within the local variable [declaration space](#_Trm00130). It is an error for a local variable [declaration space](#_Trm00130) and a nested local variable [declaration space](#_Trm00130) to contain [elements](#_Trm00094) with the same name. Thus, within a nested [declaration space](#_Trm00130) 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](#_Trm00130). It is possible for two [declaration space](#_Trm00130)s to contain [elements](#_Trm00094) with the same name as long as neither [declaration space](#_Trm00130) contains the other.
* Each [block](#_Grm00071) or [switch\_block](#_Grm00081) , as well as a *for*, *foreach* and *using* statement, creates a local variable [declaration space](#_Trm00130) for [local variables](#_Trm00067) and local constants . Names are introduced into this [declaration space](#_Trm00130) through [local\_variable\_declaration](#_Grm00076)s and [local\_constant\_declaration](#_Grm00077)s. Note that [block](#_Trm00038)s that occur as or within the body of a function member or anonymous function are nested within the local variable [declaration space](#_Trm00130) declared by those functions for their [parameters](#_Trm00059). Thus it is an error to have e.g. a [method](#_Trm00056) with a local variable and a parameter of the same name.
* Each [block](#_Grm00071) or [switch\_block](#_Grm00081) creates a separate [declaration space](#_Trm00130) for labels. Names are introduced into this [declaration space](#_Trm00130) through [labeled\_statement](#_Grm00074)s, and the names are referenced through [goto\_statement](#_Grm00090)s. The ***label declaration space*** of a [block](#_Trm00038) includes any nested [block](#_Trm00038)s. Thus, within a nested [block](#_Trm00038) it is not possible to declare a label with the same name as a label in an enclosing [block](#_Trm00038).

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](#_Trm00010), constants, [method](#_Trm00056)s, properties, [event](#_Trm00088)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance constructor](#_Trm00080)s, [destructor](#_Trm00091)s, [static constructor](#_Trm00081)s, and [types](#_Trm00011). 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](#_Trm00031) must be [defined](#_Trm00121) before they are used ([§3.7](#_Toc00082)).
* Declaration order for enum member declarations ([§14.3](#_Toc00554)) is significant when [constant\_expression](#_Grm00068) values are omitted.

The [declaration space](#_Trm00130) of a namespace is "open ended", and two namespace declarations with the same fully qualified name contribute to the same [declaration space](#_Trm00130). For example

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

The two namespace declarations above contribute to the same [declaration space](#_Trm00130), 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](#_Trm00130), 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](#_Trm00130) of a [block](#_Trm00038) includes any nested [block](#_Trm00038)s. Thus, in the following example, the F and G [method](#_Trm00056)s result in a compile-time error because the name i is declared in the outer [block](#_Trm00038) and cannot be redeclared in the inner [block](#_Trm00038). However, the H and I [method](#_Trm00056)s are valid since the two i's are declared in separate non-nested [block](#_Trm00038)s.

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](#_Trm00011) have ***members***. The [members](#_Trm00012) 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](#_Trm00012) of the base class, except [instance constructor](#_Trm00080)s, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s, become [members](#_Trm00012) of the derived type. The declared accessibility of a base class member does not control whether the member is [inherited](#_Trm00136)—[inheritance](#_Trm00047) extends to any member that isn't an [instance constructor](#_Trm00080), [static constructor](#_Trm00081), or [destructor](#_Trm00091). However, an [inherited](#_Trm00136) member may not be accessible in a derived type, either because of its declared accessibility ([§3.5.1](#_Toc00077)) or because it is hidden by a declaration in the type itself ([§3.7.1.2](#_Toc00085)).

### Namespace [members](#_Trm00012)

Namespaces and [types](#_Trm00011) that have no enclosing namespace are [members](#_Trm00012) of the ***global namespace***. This corresponds directly to the names declared in the global [declaration space](#_Trm00130).

Namespaces and [types](#_Trm00011) declared within a namespace are [members](#_Trm00012) of that namespace. This corresponds directly to the names declared in the [declaration space](#_Trm00130) of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal [namespaces](#_Trm00010), and namespace names are always publicly accessible.

### Struct [members](#_Trm00012)

The [members](#_Trm00012) of a struct are the [members](#_Trm00012) declared in the struct and the [members](#_Trm00012) [inherited](#_Trm00136) from the struct's direct base class System.ValueType and the indirect base class object.

The [members](#_Trm00012) of a simple type correspond directly to the [members](#_Trm00012) of the struct type aliased by the simple type:

* The [members](#_Trm00012) of sbyte are the [members](#_Trm00012) of the System.SByte struct.
* The [members](#_Trm00012) of byte are the [members](#_Trm00012) of the System.Byte struct.
* The [members](#_Trm00012) of short are the [members](#_Trm00012) of the System.Int16 struct.
* The [members](#_Trm00012) of ushort are the [members](#_Trm00012) of the System.UInt16 struct.
* The [members](#_Trm00012) of int are the [members](#_Trm00012) of the System.Int32 struct.
* The [members](#_Trm00012) of uint are the [members](#_Trm00012) of the System.UInt32 struct.
* The [members](#_Trm00012) of long are the [members](#_Trm00012) of the System.Int64 struct.
* The [members](#_Trm00012) of ulong are the [members](#_Trm00012) of the System.UInt64 struct.
* The [members](#_Trm00012) of char are the [members](#_Trm00012) of the System.Char struct.
* The [members](#_Trm00012) of float are the [members](#_Trm00012) of the System.Single struct.
* The [members](#_Trm00012) of double are the [members](#_Trm00012) of the System.Double struct.
* The [members](#_Trm00012) of decimal are the [members](#_Trm00012) of the System.Decimal struct.
* The [members](#_Trm00012) of bool are the [members](#_Trm00012) of the System.Boolean struct.

### Enumeration [members](#_Trm00012)

The [members](#_Trm00012) of an enumeration are the constants declared in the enumeration and the [members](#_Trm00012) [inherited](#_Trm00136) from the enumeration's direct base class System.Enum and the indirect [base classes](#_Trm00050) System.ValueType and object.

### Class [members](#_Trm00012)

The [members](#_Trm00012) of a class are the [members](#_Trm00012) declared in the class and the [members](#_Trm00012) [inherited](#_Trm00136) from the base class (except for class object which has no base class). The [members](#_Trm00012) [inherited](#_Trm00136) from the base class include the constants, fields, [method](#_Trm00056)s, properties, [event](#_Trm00088)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, and [types](#_Trm00011) of the base class, but not the [instance constructor](#_Trm00080)s, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s of the base class. Base class [members](#_Trm00012) are [inherited](#_Trm00136) without regard to their accessibility.

A class declaration may contain declarations of constants, fields, [method](#_Trm00056)s, properties, [event](#_Trm00088)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance constructor](#_Trm00080)s, [destructor](#_Trm00091)s, [static constructor](#_Trm00081)s and [types](#_Trm00011).

The [members](#_Trm00012) of object and string correspond directly to the [members](#_Trm00012) of the [class types](#_Trm00024) they alias:

* The [members](#_Trm00012) of object are the [members](#_Trm00012) of the System.Object class.
* The [members](#_Trm00012) of string are the [members](#_Trm00012) of the System.String class.

### Interface [members](#_Trm00012)

The [members](#_Trm00012) of an [interface](#_Trm00102) are the [members](#_Trm00012) declared in the [interface](#_Trm00102) and in all base [interface](#_Trm00102)s of the [interface](#_Trm00102). The [members](#_Trm00012) in class object are not, strictly speaking, [members](#_Trm00012) of any [interface](#_Trm00102) ([§13.2](#_Toc00536)). However, the [members](#_Trm00012) in class object are available via member lookup in any [interface](#_Trm00102) type ([§7.4](#_Toc00221)).

### Array [members](#_Trm00012)

The [members](#_Trm00012) of an [array](#_Trm00093) are the [members](#_Trm00012) [inherited](#_Trm00136) from class System.Array.

### Delegate [members](#_Trm00012)

The [members](#_Trm00012) of a delegate are the [members](#_Trm00012) [inherited](#_Trm00136) from class System.Delegate.

## Member access

Declarations of [members](#_Trm00012) allow control over member access. The accessibility of a member is established by the declared accessibility ([§3.5.1](#_Toc00077)) 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 ([§3.5.2](#_Toc00078)) 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](#_Trm00011) 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](#_Trm00109)".
* 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](#_Trm00109) or [types](#_Trm00011) 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](#_Trm00011) of [declared accessibility](#_Trm00140) 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](#_Trm00140).

* Namespaces implicitly have public [declared accessibility](#_Trm00140). No access modifiers are allowed on namespace declarations.
* Types declared in [compilation units](#_Trm00111) or [namespaces](#_Trm00010) can have public or internal [declared accessibility](#_Trm00140) and default to internal [declared accessibility](#_Trm00140).
* Class [members](#_Trm00012) can have any of the five kinds of [declared accessibility](#_Trm00140) and default to private [declared accessibility](#_Trm00140). (Note that a type declared as a member of a class can have any of the five kinds of [declared accessibility](#_Trm00140), whereas a type declared as a member of a namespace can have only public or internal [declared accessibility](#_Trm00140).)
* Struct [members](#_Trm00012) can have public, internal, or private [declared accessibility](#_Trm00140) and default to private [declared accessibility](#_Trm00140) because [structs](#_Trm00092) are implicitly sealed. Struct [members](#_Trm00012) introduced in a struct (that is, not [inherited](#_Trm00136) by that struct) cannot have protected or protected internal [declared accessibility](#_Trm00140). (Note that a type declared as a member of a struct can have public, internal, or private [declared accessibility](#_Trm00140), whereas a type declared as a member of a namespace can have only public or internal [declared accessibility](#_Trm00140).)
* Interface [members](#_Trm00012) implicitly have public [declared accessibility](#_Trm00140). No access modifiers are allowed on [interface](#_Trm00102) member declarations.
* Enumeration [members](#_Trm00012) implicitly have public [declared accessibility](#_Trm00140). 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](#_Trm00109) text in which access to the member is permitted. For purposes of defining the [accessibility domain](#_Trm00141) 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](#_Trm00109) is [defined](#_Trm00121) as all [program](#_Trm00109) text contained in all [source files](#_Trm00110) of the [program](#_Trm00109), and the [program](#_Trm00109) text of a type is [defined](#_Trm00121) as all [program](#_Trm00109) text contained in the [type\_declaration](#_Grm00105)s of that type (including, possibly, [types](#_Trm00011) that are [nested](#_Trm00143) within the type).

The [accessibility domain](#_Trm00141) of a pre[defined](#_Trm00121) type (such as object, int, or double) is unlimited.

The [accessibility domain](#_Trm00141) of a [top-level](#_Trm00142) unbound type T ([§4.4.3](#_Toc00116)) that is declared in a [program](#_Trm00109) P is [defined](#_Trm00121) as follows:

* If the [declared accessibility](#_Trm00140) of T is public, the [accessibility domain](#_Trm00141) of T is the [program](#_Trm00109) text of P and any [program](#_Trm00109) that references P.
* If the [declared accessibility](#_Trm00140) of T is internal, the [accessibility domain](#_Trm00141) of T is the [program](#_Trm00109) text of P.

From these definitions it follows that the [accessibility domain](#_Trm00141) of a [top-level](#_Trm00142) unbound type is always at least the [program](#_Trm00109) text of the [program](#_Trm00109) in which that type is declared.

The [accessibility domain](#_Trm00141) for a constructed type T<A1, ..., An> is the intersection of the [accessibility domain](#_Trm00141) of the unbound generic type T and the [accessibility domain](#_Trm00141)s of the type [arguments](#_Trm00062) A1, ..., An.

The [accessibility domain](#_Trm00141) of a [nested](#_Trm00143) member M declared in a type T within a [program](#_Trm00109) P is [defined](#_Trm00121) as follows (noting that M itself may possibly be a type):

* If the [declared accessibility](#_Trm00140) of M is public, the [accessibility domain](#_Trm00141) of M is the [accessibility domain](#_Trm00141) of T.
* If the [declared accessibility](#_Trm00140) of M is protected internal, let D be the union of the [program](#_Trm00109) text of P and the [program](#_Trm00109) text of any type derived from T, which is declared outside P. The [accessibility domain](#_Trm00141) of M is the intersection of the [accessibility domain](#_Trm00141) of T with D.
* If the [declared accessibility](#_Trm00140) of M is protected, let D be the union of the [program](#_Trm00109) text of T and the [program](#_Trm00109) text of any type derived from T. The [accessibility domain](#_Trm00141) of M is the intersection of the [accessibility domain](#_Trm00141) of T with D.
* If the [declared accessibility](#_Trm00140) of M is internal, the [accessibility domain](#_Trm00141) of M is the intersection of the [accessibility domain](#_Trm00141) of T with the [program](#_Trm00109) text of P.
* If the [declared accessibility](#_Trm00140) of M is private, the [accessibility domain](#_Trm00141) of M is the [program](#_Trm00109) text of T.

From these definitions it follows that the [accessibility domain](#_Trm00141) of a [nested](#_Trm00143) member is always at least the [program](#_Trm00109) text of the type in which the member is declared. Furthermore, it follows that the [accessibility domain](#_Trm00141) of a member is never more inclusive than the [accessibility domain](#_Trm00141) 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](#_Trm00138).
* 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](#_Trm00109) 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 ([§3.5.3](#_Toc00079)).
* 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 ([§3.5.3](#_Toc00079)).
* Otherwise, if M is internal, the access is permitted if it occurs within the [program](#_Trm00109) 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 in[accessible](#_Trm00138), 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](#_Trm00012) have the following [accessibility domain](#_Trm00141)s:

* The [accessibility domain](#_Trm00141) of A and A.X is unlimited.
* The [accessibility domain](#_Trm00141) of A.Y, B, B.X, B.Y, B.C, B.C.X, and B.C.Y is the [program](#_Trm00109) text of the containing [program](#_Trm00109).
* The [accessibility domain](#_Trm00141) of A.Z is the [program](#_Trm00109) text of A.
* The [accessibility domain](#_Trm00141) of B.Z and B.D is the [program](#_Trm00109) text of B, including the [program](#_Trm00109) text of B.C and B.D.
* The [accessibility domain](#_Trm00141) of B.C.Z is the [program](#_Trm00109) text of B.C.
* The [accessibility domain](#_Trm00141) of B.D.X and B.D.Y is the [program](#_Trm00109) text of B, including the [program](#_Trm00109) text of B.C and B.D.
* The [accessibility domain](#_Trm00141) of B.D.Z is the [program](#_Trm00109) text of B.D.

As the example illustrates, the [accessibility domain](#_Trm00141) of a member is never larger than that of a containing type. For example, even though all X [members](#_Trm00012) have public [declared accessibility](#_Trm00140), all but A.X have [accessibility domain](#_Trm00141)s that are constrained by a containing type.

As described in [§3.4](#_Toc00068), all [members](#_Trm00012) of a base class, except for [instance constructor](#_Trm00080)s, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s, are [inherited](#_Trm00136) by derived [types](#_Trm00011). This includes even private [members](#_Trm00012) of a base class. However, the [accessibility domain](#_Trm00141) of a private member includes only the [program](#_Trm00109) 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](#_Trm00138) within the [class\_body](#_Grm00112) of A. Thus, the access to b.x succeeds in the A.F [method](#_Trm00056), but fails in the B.F [method](#_Trm00056).

### Protected access for [instance members](#_Trm00052)

When a protected instance member is accessed outside the [program](#_Trm00109) text of the class in which it is declared, and when a protected internal instance member is accessed outside the [program](#_Trm00109) text of the [program](#_Trm00109) 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 pr[event](#_Trm00088)s one derived class from accessing protected [members](#_Trm00012) of other [derived classes](#_Trm00049), even when the [members](#_Trm00012) are [inherited](#_Trm00136) 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](#_Grm00112) of D, access to M can take one of the following forms:

* An unqualified [type\_name](#_Grm00027) or [primary\_expression](#_Grm00035) of the form M.
* A [primary\_expression](#_Grm00035) 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](#_Grm00035) of the form base.M.

In addition to these forms of access, a derived class can access a protected [instance constructor](#_Trm00080) of a base class in a [constructor\_initializer](#_Grm00123) ([§10.11.1](#_Toc00475)).

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](#_Trm00045) 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](#_Trm00045) of [class types](#_Trm00024) constructed from the generic type.

### Accessibility constraints

Several con[structs](#_Trm00092) 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](#_Trm00138) as a member or type M if the [accessibility domain](#_Trm00141) of T is a superset of the [accessibility domain](#_Trm00141) of M. In other words, T is at least as [accessible](#_Trm00138) as M if T is [accessible](#_Trm00138) in all contexts in which M is [accessible](#_Trm00138).

The following accessibility constraints exist:

* The direct base class of a class type must be at least as [accessible](#_Trm00138) as the class type itself.
* The explicit base [interface](#_Trm00102)s of an [interface](#_Trm00102) type must be at least as [accessible](#_Trm00138) as the [interface](#_Trm00102) type itself.
* The [return type](#_Trm00060) and parameter [types](#_Trm00011) of a [delegate type](#_Trm00107) must be at least as [accessible](#_Trm00138) as the [delegate type](#_Trm00107) itself.
* The type of a constant must be at least as [accessible](#_Trm00138) as the constant itself.
* The type of a field must be at least as [accessible](#_Trm00138) as the field itself.
* The [return type](#_Trm00060) and parameter [types](#_Trm00011) of a [method](#_Trm00056) must be at least as [accessible](#_Trm00138) as the [method](#_Trm00056) itself.
* The type of a property must be at least as [accessible](#_Trm00138) as the property itself.
* The type of an [event](#_Trm00088) must be at least as [accessible](#_Trm00138) as the [event](#_Trm00088) itself.
* The type and parameter [types](#_Trm00011) of an [indexer](#_Trm00087) must be at least as [accessible](#_Trm00138) as the [indexer](#_Trm00087) itself.
* The [return type](#_Trm00060) and parameter [types](#_Trm00011) of an [operator](#_Trm00090) must be at least as [accessible](#_Trm00138) as the [operator](#_Trm00090) itself.
* The parameter [types](#_Trm00011) of an [instance constructor](#_Trm00080) must be at least as [accessible](#_Trm00138) as the [instance constructor](#_Trm00080) 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](#_Trm00138) as B.

Likewise, in the example

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

the H [method](#_Trm00056) in B results in a compile-time error because the [return type](#_Trm00060) A is not at least as [accessible](#_Trm00138) as the [method](#_Trm00056).

## Signatures and [overloading](#_Trm00077)

Methods, [instance constructor](#_Trm00080)s, [indexer](#_Trm00087)s, and [operator](#_Trm00090)s are characterized by their ***signatures***:

* The [signature](#_Trm00061) of a [method](#_Trm00056) consists of the name of the [method](#_Trm00056), the number of type [parameters](#_Trm00059) and the type and kind (value, reference, or output) of each of its formal [parameters](#_Trm00059), considered in the order left to right. For these purposes, any type parameter of the [method](#_Trm00056) 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](#_Trm00056). The [signature](#_Trm00061) of a [method](#_Trm00056) specifically does not include the [return type](#_Trm00060), the params modifier that may be specified for the right-most parameter, nor the optional type parameter constraints.
* The [signature](#_Trm00061) of an [instance constructor](#_Trm00080) consists of the type and kind (value, reference, or output) of each of its formal [parameters](#_Trm00059), considered in the order left to right. The [signature](#_Trm00061) of an [instance constructor](#_Trm00080) specifically does not include the params modifier that may be specified for the right-most parameter.
* The [signature](#_Trm00061) of an [indexer](#_Trm00087) consists of the type of each of its formal [parameters](#_Trm00059), considered in the order left to right. The [signature](#_Trm00061) of an [indexer](#_Trm00087) specifically does not include the [element type](#_Trm00095), nor does it include the params modifier that may be specified for the right-most parameter.
* The [signature](#_Trm00061) of an [operator](#_Trm00090) consists of the name of the [operator](#_Trm00090) and the type of each of its formal [parameters](#_Trm00059), considered in the order left to right. The [signature](#_Trm00061) of an [operator](#_Trm00090) specifically does not include the result type.

Signatures are the enabling mechanism for ***overloading*** of [members](#_Trm00012) in classes, [structs](#_Trm00092), and [interface](#_Trm00102)s:

* Overloading of [method](#_Trm00056)s permits a class, struct, or [interface](#_Trm00102) to declare multiple [method](#_Trm00056)s with the same name, provided their [signature](#_Trm00061)s are unique within that class, struct, or [interface](#_Trm00102).
* Overloading of [instance constructor](#_Trm00080)s permits a class or struct to declare multiple [instance constructor](#_Trm00080)s, provided their [signature](#_Trm00061)s are unique within that class or struct.
* Overloading of [indexer](#_Trm00087)s permits a class, struct, or [interface](#_Trm00102) to declare multiple [indexer](#_Trm00087)s, provided their [signature](#_Trm00061)s are unique within that class, struct, or [interface](#_Trm00102).
* Overloading of [operator](#_Trm00090)s permits a class or struct to declare multiple [operator](#_Trm00090)s with the same name, provided their [signature](#_Trm00061)s are unique within that class or struct.

Although out and ref parameter modifiers are considered part of a [signature](#_Trm00061), [members](#_Trm00012) declared in a single type cannot differ in [signature](#_Trm00061) solely by ref and out. A compile-time error occurs if two [members](#_Trm00012) are declared in the same type with [signature](#_Trm00061)s that would be the same if all [parameters](#_Trm00059) in both [method](#_Trm00056)s with out modifiers were changed to ref modifiers. For other purposes of [signature](#_Trm00061) matching (e.g., hiding or overriding), ref and out are considered part of the [signature](#_Trm00061) and do not match each other. (This restriction is to allow C#  [program](#_Trm00109)s to be easily translated to run on the Common Language Infrastructure (CLI), which does not provide a way to define [method](#_Trm00056)s that differ solely in ref and out.)

For the purposes of singatures, the [types](#_Trm00011) object and dynamic are considered the same. Members declared in a single type can therefore not differ in [signature](#_Trm00061) solely by object and dynamic.

The following example shows a set of [overloaded](#_Trm00036) [method](#_Trm00056) declarations along with their [signature](#_Trm00061)s.

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 ([§10.6.1](#_Toc00442)) are part of a [signature](#_Trm00061). Thus, F(int) and F(ref int) are unique [signature](#_Trm00061)s. However, F(ref int) and F(out int) cannot be declared within the same [interface](#_Trm00102) because their [signature](#_Trm00061)s differ solely by ref and out. Also, note that the [return type](#_Trm00060) and the params modifier are not part of a [signature](#_Trm00061), so it is not possible to overload solely based on [return type](#_Trm00060) or on the inclusion or exclusion of the params modifier. As such, the declarations of the [method](#_Trm00056)s 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](#_Trm00109) 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](#_Trm00148) may redeclare the meaning of a name from an outer [scope](#_Trm00148) (this does not, however, remove the restriction imposed by [§3.3](#_Toc00067) that within a [nested](#_Trm00143) [block](#_Trm00038) it is not possible to declare a local variable with the same name as a local variable in an enclosing [block](#_Trm00038)). The name from the outer [scope](#_Trm00148) is then said to be ***hidden*** in the region of [program](#_Trm00109) text covered by the inner [scope](#_Trm00148), and access to the outer name is only possible by qualifying the name.

* The [scope](#_Trm00148) of a namespace member declared by a [namespace\_member\_declaration](#_Grm00104) ([§9.5](#_Toc00384)) with no enclosing [namespace\_declaration](#_Grm00099) is the entire [program](#_Trm00109) text.
* The [scope](#_Trm00148) of a namespace member declared by a [namespace\_member\_declaration](#_Grm00104) within a [namespace\_declaration](#_Grm00099) whose fully qualified name is N is the [namespace\_body](#_Grm00099) of every [namespace\_declaration](#_Grm00099) whose fully qualified name is N or starts with N, followed by a period.
* The [scope](#_Trm00148) of name [defined](#_Trm00121) by an [extern\_alias\_directive](#_Grm00100) extends over the [using\_directive](#_Grm00101)s, [global\_attributes](#_Grm00147) and [namespace\_member\_declaration](#_Grm00104)s of its immediately containing compilation unit or namespace body. An [extern\_alias\_directive](#_Grm00100) does not contribute any new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130). In other words, an [extern\_alias\_directive](#_Grm00100) is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
* The [scope](#_Trm00148) of a name [defined](#_Trm00121) or imported by a [using\_directive](#_Grm00101) ([§9.4](#_Toc00381)) extends over the [namespace\_member\_declaration](#_Grm00104)s of the [compilation\_unit](#_Grm00098) or [namespace\_body](#_Grm00099) in which the [using\_directive](#_Grm00101) occurs. A [using\_directive](#_Grm00101) may make zero or more namespace or type names available within a particular [compilation\_unit](#_Grm00098) or [namespace\_body](#_Grm00099), but does not contribute any new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130). In other words, a [using\_directive](#_Grm00101) is not transitive but rather affects only the [compilation\_unit](#_Grm00098) or [namespace\_body](#_Grm00099) in which it occurs.
* The [scope](#_Trm00148) of a type parameter declared by a [type\_parameter\_list](#_Grm00109) on a [class\_declaration](#_Grm00107) ([§10.1](#_Toc00389)) is the [class\_base](#_Grm00110), [type\_parameter\_constraints\_clause](#_Grm00111)s, and [class\_body](#_Grm00112) of that [class\_declaration](#_Grm00107).
* The [scope](#_Trm00148) of a type parameter declared by a [type\_parameter\_list](#_Grm00109) on a [struct\_declaration](#_Grm00126) ([§11.1](#_Toc00498)) is the [struct\_interfaces](#_Grm00128), [type\_parameter\_constraints\_clause](#_Grm00111)s, and [struct\_body](#_Grm00129) of that [struct\_declaration](#_Grm00126).
* The [scope](#_Trm00148) of a type parameter declared by a [type\_parameter\_list](#_Grm00109) on an [interface\_declaration](#_Grm00133) ([§13.1](#_Toc00528)) is the [interface\_base](#_Grm00136), [type\_parameter\_constraints\_clause](#_Grm00111)s, and [interface\_body](#_Grm00137) of that [interface\_declaration](#_Grm00133).
* The [scope](#_Trm00148) of a type parameter declared by a [type\_parameter\_list](#_Grm00109) on a [delegate\_declaration](#_Grm00146) ([§15.1](#_Toc00558)) is the [return\_type](#_Grm00116), [formal\_parameter\_list](#_Grm00117), and [type\_parameter\_constraints\_clause](#_Grm00111)s of that [delegate\_declaration](#_Grm00146).
* The [scope](#_Trm00148) of a member declared by a [class\_member\_declaration](#_Grm00113) ([§10.1.6](#_Toc00400)) is the [class\_body](#_Grm00112) in which the declaration occurs. In addition, the [scope](#_Trm00148) of a class member extends to the [class\_body](#_Grm00112) of those [derived classes](#_Trm00049) that are included in the [accessibility domain](#_Trm00141) ([§3.5.2](#_Toc00078)) of the member.
* The [scope](#_Trm00148) of a member declared by a [struct\_member\_declaration](#_Grm00130) ([§11.2](#_Toc00503)) is the [struct\_body](#_Grm00129) in which the declaration occurs.
* The [scope](#_Trm00148) of a member declared by an [enum\_member\_declaration](#_Grm00145) ([§14.3](#_Toc00554)) is the [enum\_body](#_Grm00143) in which the declaration occurs.
* The [scope](#_Trm00148) of a parameter declared in a [method\_declaration](#_Grm00116) ([§10.6](#_Toc00441)) is the [method\_body](#_Grm00116) of that [method\_declaration](#_Grm00116).
* The [scope](#_Trm00148) of a parameter declared in an [indexer\_declaration](#_Grm00121) ([§10.9](#_Toc00468)) is the [accessor\_declarations](#_Grm00119) of that [indexer\_declaration](#_Grm00121).
* The [scope](#_Trm00148) of a parameter declared in an [operator\_declaration](#_Grm00122) ([§10.10](#_Toc00470)) is the [block](#_Grm00071) of that [operator\_declaration](#_Grm00122).
* The [scope](#_Trm00148) of a parameter declared in a [constructor\_declaration](#_Grm00123) ([§10.11](#_Toc00474)) is the [constructor\_initializer](#_Grm00123) and [block](#_Grm00071) of that [constructor\_declaration](#_Grm00123).
* The [scope](#_Trm00148) of a parameter declared in a [lambda\_expression](#_Grm00064) ([§7.15](#_Toc00321)) is the [anonymous\_function\_body](#_Grm00064) of that [lambda\_expression](#_Grm00064)
* The [scope](#_Trm00148) of a parameter declared in an [anonymous\_method\_expression](#_Grm00064) ([§7.15](#_Toc00321)) is the [block](#_Grm00071) of that [anonymous\_method\_expression](#_Grm00064).
* The [scope](#_Trm00148) of a label declared in a [labeled\_statement](#_Grm00074) ([§8.4](#_Toc00353)) is the [block](#_Grm00071) in which the declaration occurs.
* The [scope](#_Trm00148) of a local variable declared in a [local\_variable\_declaration](#_Grm00076) ([§8.5.1](#_Toc00355)) is the [block](#_Trm00038) in which the declaration occurs.
* The [scope](#_Trm00148) of a local variable declared in a [switch\_block](#_Grm00081) of a switch statement ([§8.7.2](#_Toc00360)) is the [switch\_block](#_Grm00081).
* The [scope](#_Trm00148) of a local variable declared in a [for\_initializer](#_Grm00085) of a for statement ([§8.8.3](#_Toc00364)) is the [for\_initializer](#_Grm00085), the [for\_condition](#_Grm00085), the [for\_iterator](#_Grm00085), and the contained [statement](#_Grm00070) of the for statement.
* The [scope](#_Trm00148) of a local constant declared in a [local\_constant\_declaration](#_Grm00077) ([§8.5.2](#_Toc00356)) is the [block](#_Trm00038) 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](#_Grm00077).
* The [scope](#_Trm00148) of a variable declared as part of a [foreach\_statement](#_Grm00086), [using\_statement](#_Grm00096), [lock\_statement](#_Grm00095) or [query\_expression](#_Grm00065) is determined by the expansion of the given construct.

Within the [scope](#_Trm00148) 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](#_Trm00148) 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](#_Grm00076) 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](#_Trm00056) above, the first assignment to i specifically does not refer to the field declared in the outer [scope](#_Trm00148). 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](#_Trm00056), the use of j in the initializer for the declaration of j is valid because the use does not precede the [local\_variable\_declarator](#_Grm00076). In the H [method](#_Trm00056), a subsequent [local\_variable\_declarator](#_Grm00076) correctly refers to a local variable declared in an earlier [local\_variable\_declarator](#_Grm00076) within the same [local\_variable\_declaration](#_Grm00076).

The scoping rules for [local variables](#_Trm00067) are designed to guarantee that the meaning of a name used in an expression context is always the same within a [block](#_Trm00038). If the [scope](#_Trm00148) of a local variable were to extend only from its declaration to the end of the [block](#_Trm00038), 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](#_Trm00037) of the [block](#_Trm00038) were later to be rearranged.

The meaning of a name within a [block](#_Trm00038) 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](#_Trm00148) of an entity typically encompasses more [program](#_Trm00109) text than the [declaration space](#_Trm00130) of the entity. In particular, the [scope](#_Trm00148) of an entity may include declarations that introduce new [declaration space](#_Trm00130)s 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](#_Trm00150).

Name hiding occurs when [scope](#_Trm00148)s overlap through nesting and when [scope](#_Trm00148)s overlap through [inheritance](#_Trm00047). The characteristics of the two [types](#_Trm00011) of hiding are described in the following sections.

#### Hiding through nesting

Name hiding through nesting can occur as a result of nesting [namespaces](#_Trm00010) or [types](#_Trm00011) within [namespaces](#_Trm00010), as a result of nesting [types](#_Trm00011) within classes or [structs](#_Trm00092), 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](#_Trm00056), the instance variable i is [hidden](#_Trm00150) by the local variable i, but within the G [method](#_Trm00056), i still refers to the instance variable.

When a name in an inner [scope](#_Trm00148) [hide](#_Trm00132)s a name in an outer [scope](#_Trm00148), it [hide](#_Trm00132)s all [overloaded](#_Trm00036) 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](#_Trm00150) by the inner declaration. For the same reason, the call F("Hello") results in a compile-time error.

#### Hiding through [inheritance](#_Trm00047)

Name hiding through [inheritance](#_Trm00047) occurs when classes or [structs](#_Trm00092) redeclare names that were [inherited](#_Trm00136) from [base classes](#_Trm00050). This type of name hiding takes one of the following forms:

* A constant, field, property, [event](#_Trm00088), or type introduced in a class or struct [hide](#_Trm00132)s all base class [members](#_Trm00012) with the same name.
* A [method](#_Trm00056) introduced in a class or struct [hide](#_Trm00132)s all non-[method](#_Trm00056) base class [members](#_Trm00012) with the same name, and all base class [method](#_Trm00056)s with the same [signature](#_Trm00061) ([method](#_Trm00056) name and parameter count, modifiers, and [types](#_Trm00011)).
* An [indexer](#_Trm00087) introduced in a class or struct [hide](#_Trm00132)s all base class [indexer](#_Trm00087)s with the same [signature](#_Trm00061) (parameter count and [types](#_Trm00011)).

The rules governing [operator](#_Trm00090) declarations ([§10.10](#_Toc00470)) make it impossible for a derived class to declare an [operator](#_Trm00090) with the same [signature](#_Trm00061) as an [operator](#_Trm00090) in a base class. Thus, [operator](#_Trm00090)s never [hide](#_Trm00132) one another.

Contrary to hiding a name from an outer [scope](#_Trm00148), hiding an [accessible](#_Trm00138) name from an [inherited](#_Trm00136) [scope](#_Trm00148) 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](#_Trm00136) name is specifically not an error, since that would preclude separate evolution of [base classes](#_Trm00050). For example, the above situation might have come about because a later version of Base introduced an F [method](#_Trm00056) 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](#_Trm00049) to become invalid.

The warning caused by hiding an [inherited](#_Trm00136) 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](#_Trm00132) the [inherited](#_Trm00136) member.

A declaration of a new member [hide](#_Trm00132)s an [inherited](#_Trm00136) member only within the [scope](#_Trm00148) 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 [hide](#_Trm00132)s the F that was [inherited](#_Trm00136) from Base, but since the new F in Derived has private access, its [scope](#_Trm00148) 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](#_Trm00109) require a [namespace\_name](#_Grm00027) or a [type\_name](#_Grm00027) 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](#_Grm00027) is a [namespace\_or\_type\_name](#_Grm00027) that refers to a namespace. Following resolution as described below, the [namespace\_or\_type\_name](#_Grm00027) of a [namespace\_name](#_Grm00027) must refer to a namespace, or otherwise a compile-time error occurs. No type [arguments](#_Trm00062) ([§4.4.1](#_Toc00114)) can be present in a [namespace\_name](#_Grm00027) (only [types](#_Trm00011) can have type [arguments](#_Trm00062)).

A [type\_name](#_Grm00027) is a [namespace\_or\_type\_name](#_Grm00027) that refers to a type. Following resolution as described below, the [namespace\_or\_type\_name](#_Grm00027) of a [type\_name](#_Grm00027) must refer to a type, or otherwise a compile-time error occurs.

If the [namespace\_or\_type\_name](#_Grm00027) is a qualified-alias-member its meaning is as described in [§9.7](#_Toc00386). Otherwise, a [namespace\_or\_type\_name](#_Grm00027) 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](#_Grm00027) and <A1, ..., Ak> is an optional [type\_argument\_list](#_Grm00031). When no [type\_argument\_list](#_Grm00031) is specified, consider k to be zero.

The meaning of a [namespace\_or\_type\_name](#_Grm00027) is determined as follows:

* If the [namespace\_or\_type\_name](#_Grm00027) is of the form I or of the form I<A1, ..., Ak>:
  + If K is zero and the [namespace\_or\_type\_name](#_Grm00027) appears within a generic [method](#_Trm00056) declaration ([§10.6](#_Toc00441)) and if that declaration includes a type parameter ([§10.1.3](#_Toc00395)) with name I, then the [namespace\_or\_type\_name](#_Grm00027) refers to that type parameter.
  + Otherwise, if the [namespace\_or\_type\_name](#_Grm00027) appears within a type declaration, then for each instance type T ([§10.3.1](#_Toc00411)), 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](#_Grm00027) refers to that type parameter.
    - Otherwise, if the [namespace\_or\_type\_name](#_Grm00027) appears within the body of the type declaration, and T or any of its base [types](#_Trm00011) contain a [nested](#_Trm00143) [accessible](#_Trm00138) type having name I and K type [parameters](#_Trm00059), then the [namespace\_or\_type\_name](#_Grm00027) refers to that type constructed with the given type [arguments](#_Trm00062). If there is more than one such type, the type declared within the more derived type is selected. Note that non-type [members](#_Trm00012) (constants, fields, [method](#_Trm00056)s, properties, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance constructor](#_Trm00080)s, [destructor](#_Trm00091)s, and [static constructor](#_Trm00081)s) and type [members](#_Trm00012) with a different number of type [parameters](#_Trm00059) are ignored when determining the meaning of the [namespace\_or\_type\_name](#_Grm00027).
  + If the previous steps were unsuccessful then, for each namespace N, starting with the namespace in which the [namespace\_or\_type\_name](#_Grm00027) occurs, continuing with each enclosing namespace (if any), and ending with the [global namespace](#_Trm00137), 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](#_Grm00027) occurs is enclosed by a namespace declaration for N and the namespace declaration contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) that associates the name I with a namespace or type, then the [namespace\_or\_type\_name](#_Grm00027) is ambiguous and a compile-time error occurs.
      * Otherwise, the [namespace\_or\_type\_name](#_Grm00027) refers to the namespace named I in N.
    - Otherwise, if N contains an [accessible](#_Trm00138) type having name I and K type [parameters](#_Trm00059), then:
      * If K is zero and the location where the [namespace\_or\_type\_name](#_Grm00027) occurs is enclosed by a namespace declaration for N and the namespace declaration contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) that associates the name I with a namespace or type, then the [namespace\_or\_type\_name](#_Grm00027) is ambiguous and a compile-time error occurs.
      * Otherwise, the [namespace\_or\_type\_name](#_Grm00027) refers to the type constructed with the given type [arguments](#_Trm00062).
    - Otherwise, if the location where the [namespace\_or\_type\_name](#_Grm00027) occurs is enclosed by a namespace declaration for N:
      * If K is zero and the namespace declaration contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) that associates the name I with an imported namespace or type, then the [namespace\_or\_type\_name](#_Grm00027) refers to that namespace or type.
      * Otherwise, if the [namespaces](#_Trm00010) imported by the [using\_namespace\_directive](#_Grm00103)s of the namespace declaration contain exactly one type having name I and K type [parameters](#_Trm00059), then the [namespace\_or\_type\_name](#_Grm00027) refers to that type constructed with the given type [arguments](#_Trm00062).
      * Otherwise, if the [namespaces](#_Trm00010) imported by the [using\_namespace\_directive](#_Grm00103)s of the namespace declaration contain more than one type having name I and K type [parameters](#_Trm00059), then the [namespace\_or\_type\_name](#_Grm00027) is ambiguous and an error occurs.
  + Otherwise, the [namespace\_or\_type\_name](#_Grm00027) is un[defined](#_Trm00121) and a compile-time error occurs.
* Otherwise, the [namespace\_or\_type\_name](#_Grm00027) is of the form N.I or of the form N.I<A1, ..., Ak>. N is first resolved as a [namespace\_or\_type\_name](#_Grm00027). 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](#_Trm00143) namespace with name I, then the [namespace\_or\_type\_name](#_Grm00027) refers to that [nested](#_Trm00143) namespace.
  + Otherwise, if N refers to a namespace and N contains an [accessible](#_Trm00138) type having name I and K type [parameters](#_Trm00059), then the [namespace\_or\_type\_name](#_Grm00027) refers to that type constructed with the given type [arguments](#_Trm00062).
  + Otherwise, if N refers to a (possibly constructed) class or struct type and N or any of its [base classes](#_Trm00050) contain a [nested](#_Trm00143) [accessible](#_Trm00138) type having name I and K type [parameters](#_Trm00059), then the [namespace\_or\_type\_name](#_Grm00027) refers to that type constructed with the given type [arguments](#_Trm00062). 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 ([§10.1.4.1](#_Toc00397)).
  + Otherwise, N.I is an invalid [namespace\_or\_type\_name](#_Grm00027), and a compile-time error occurs.

A [namespace\_or\_type\_name](#_Grm00027) is permitted to reference a static class ([§10.1.1.3](#_Toc00393)) only if

* The [namespace\_or\_type\_name](#_Grm00027) is the T in a [namespace\_or\_type\_name](#_Grm00027) of the form T.I, or
* The [namespace\_or\_type\_name](#_Grm00027) is the T in a [typeof\_expression](#_Grm00050) ([§7.5.1](#_Toc00224)1) 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](#_Trm00153) of a namespace or type N is determined as follows:

* If N is a member of the [global namespace](#_Trm00137), its [fully qualified name](#_Trm00153) is N.
* Otherwise, its [fully qualified name](#_Trm00153) is S.N, where S is the [fully qualified name](#_Trm00153) of the namespace or type in which N is declared.

In other words, the [fully qualified name](#_Trm00153) of N is the complete hierarchical path of identifiers that lead to N, starting from the [global namespace](#_Trm00137). Because every member of a namespace or type must have a unique name, it follows that the [fully qualified name](#_Trm00153) of a namespace or type is always unique.

The example below shows several namespace and [type declarations](#_Trm00028) along with their associated [fully qualified name](#_Trm00153)s.

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](#_Trm00046). 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 [destructor](#_Trm00091)s, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the [garbage collector](#_Trm00154) 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](#_Trm00148) 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](#_Trm00154) 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](#_Trm00091) ([§10.13](#_Toc00482)) (if any) for the object is run. Under normal circumstances the [destructor](#_Trm00091) for the object is run once only, though implementation-specific APIs may allow this behavior to be [overridden](#_Trm00075).
4. Once the [destructor](#_Trm00091) 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 [destructor](#_Trm00091)s, the object is considered in[accessible](#_Trm00138) and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the [garbage collector](#_Trm00154) frees the memory associated with that object.

The [garbage collector](#_Trm00154) 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 in[accessible](#_Trm00138).

Like other languages that assume the existence of a [garbage collector](#_Trm00154), C# is designed so that the [garbage collector](#_Trm00154) may implement a wide range of memory management policies. For instance, C# does not require that [destructor](#_Trm00091)s be run or that [objects](#_Trm00046) be collected as soon as they are eligible, or that [destructor](#_Trm00091)s be run in any particular order, or on any particular thread.

The behavior of the [garbage collector](#_Trm00154) can be controlled, to some degree, via static [method](#_Trm00056)s on the class System.GC. This class can be used to request a collection to occur, [destructor](#_Trm00091)s to be run (or not run), and so forth.

Since the [garbage collector](#_Trm00154) is allowed wide latitude in deciding when to collect [objects](#_Trm00046) and run [destructor](#_Trm00091)s, a conforming implementation may produce output that differs from that shown by the following code. The [program](#_Trm00109)

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](#_Trm00046) 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](#_Trm00046) 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](#_Trm00109), if the [garbage collector](#_Trm00154) chooses to run the [destructor](#_Trm00091) of A before the [destructor](#_Trm00091) of B, then the output of this [program](#_Trm00109) 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](#_Trm00091) was run, it is still possible for [method](#_Trm00056)s of A (in this case, F) to be called from another [destructor](#_Trm00091). Also, note that running of a [destructor](#_Trm00091) may cause an object to become usable from the mainline [program](#_Trm00109) again. In this case, the running of B's [destructor](#_Trm00091) caused an instance of A that was previously not in use to become [accessible](#_Trm00138) 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 [destructor](#_Trm00091)s to only perform cleanup on data stored in their object's own fields, and not to perform any actions on referenced [objects](#_Trm00046) or [static field](#_Trm00053)s.

An alternative to using [destructor](#_Trm00091)s is to let a class implement the System.IDisposable [interface](#_Trm00102). 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 ([§8.13](#_Toc00375)).

## Execution order

Execution of a C# [program](#_Trm00109) proceeds such that the side effects of each executing thread are preserved at critical execution points. A ***side effect*** is [defined](#_Trm00121) 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 effect](#_Trm00155)s must be preserved are references to volatile fields ([§10.5.3](#_Toc00436)), lock [statements](#_Trm00037) ([§8.12](#_Toc00374)), and thread creation and termination. The execution environment is free to change the order of execution of a C# [program](#_Trm00109), 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](#_Trm00037) in the thread were executed in original [program](#_Trm00109) order.
* Initialization ordering rules are preserved ([§10.5.4](#_Toc00437) and [§10.5.5](#_Toc00438)).
* The ordering of [side effect](#_Trm00155)s is preserved with respect to volatile reads and writes ([§10.5.3](#_Toc00436)). 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 effect](#_Trm00155)s are produced (including any caused by calling a [method](#_Trm00056) or accessing a volatile field). When [program](#_Trm00109) execution is interrupted by an asynchronous [event](#_Trm00088) (such as an exception thrown by another thread), it is not guaranteed that the observable [side effect](#_Trm00155)s are [visible](#_Trm00152) in the original [program](#_Trm00109) order.

# Types

The [types](#_Trm00011) of the C# language are divided into two main categories: ***value types*** and ***reference types***. Both value [types](#_Trm00011) and [reference types](#_Trm00019) may be ***generic types***, which take one or more ***type parameters***. Type [parameters](#_Trm00059) can designate both value [types](#_Trm00011) and [reference types](#_Trm00019).

type:  
 | value\_type  
 | reference\_type  
 | type\_parameter  
 | type\_unsafe  
 ;

The final category of [types](#_Trm00011), pointers, is available only in unsafe code. This is discussed further in [§18.2](#_Toc00592).

Value [types](#_Trm00011) differ from [reference types](#_Trm00019) in that [variables](#_Trm00031) of the value [types](#_Trm00011) directly contain their data, whereas [variables](#_Trm00031) of the [reference types](#_Trm00019) store ***references*** to their data, the latter being known as ***objects***. With [reference types](#_Trm00019), it is possible for two [variables](#_Trm00031) 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](#_Trm00011), the [variables](#_Trm00031) 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](#_Trm00011). Values of [reference types](#_Trm00019) are treated as [objects](#_Trm00046) simply by viewing the values as type object. Values of value [types](#_Trm00011) are treated as [objects](#_Trm00046) by performing [boxing](#_Trm00029) and un[boxing](#_Trm00029) operations ([§4.3](#_Toc00110)).

## Value [types](#_Trm00011)

A value type is either a struct type or an enumeration type. C# provides a set of pre[defined](#_Trm00121) [struct types](#_Trm00022) called the ***simple types***. The [simple types](#_Trm00020) 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](#_Trm00011) 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](#_Trm00011) are thus implicitly sealed ([§10.1.1.2](#_Toc00392)).

Note that System.ValueType is not itself a [value\_type](#_Grm00029). Rather, it is a [class\_type](#_Grm00030) from which all [value\_type](#_Grm00029)s are automatically derived.

### Default constructors

All value [types](#_Trm00011) implicitly declare a public parameterless [instance constructor](#_Trm00080) called the ***default constructor***. The [default constructor](#_Trm00163) returns a zero-initialized instance known as the ***default value*** for the value type:

* For all [simple\_type](#_Grm00029)s, the [default value](#_Trm00164) is the value produced by a bit pattern of all zeros:
  + For sbyte, byte, short, ushort, int, uint, long, and ulong, the [default value](#_Trm00164) is 0.
  + For char, the [default value](#_Trm00164) is '\x0000'.
  + For float, the [default value](#_Trm00164) is 0.0f.
  + For double, the [default value](#_Trm00164) is 0.0d.
  + For decimal, the [default value](#_Trm00164) is 0.0m.
  + For bool, the [default value](#_Trm00164) is false.
* For an [enum\_type](#_Grm00029) E, the [default value](#_Trm00164) is 0, converted to the type E.
* For a [struct\_type](#_Grm00029), the [default value](#_Trm00164) is the value produced by setting all value type fields to their [default value](#_Trm00164) and all reference type fields to null.
* For a [nullable\_type](#_Grm00029) the [default value](#_Trm00164) is an instance for which the HasValue property is false and the Value property is un[defined](#_Trm00121). The [default value](#_Trm00164) is also known as the ***null value*** of the nullable type.

Like any other [instance constructor](#_Trm00080), the [default constructor](#_Trm00163) of a value type is invoked using the new [operator](#_Trm00090). For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. In the example below, [variables](#_Trm00031) 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](#_Trm00080), 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 constructor](#_Trm00080)s ([§11.3.8](#_Toc00512)).

### Struct [types](#_Trm00011)

A struct type is a value type that can declare constants, fields, [method](#_Trm00056)s, properties, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance constructor](#_Trm00080)s, [static constructor](#_Trm00081)s, and [nested](#_Trm00143) [types](#_Trm00011). The declaration of [struct types](#_Trm00022) is described in [§11.1](#_Toc00498).

### Simple [types](#_Trm00011)

C# provides a set of pre[defined](#_Trm00121) [struct types](#_Trm00022) called the ***simple types***. The [simple types](#_Trm00020) are identified through reserved words, but these reserved words are simply aliases for pre[defined](#_Trm00121) [struct types](#_Trm00022) in the System namespace, as described in the table below.

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

Because a simple type aliases a struct type, every simple type has [members](#_Trm00012). For example, int has the [members](#_Trm00012) declared in System.Int32 and the [members](#_Trm00012) [inherited](#_Trm00136) from System.Object, and the following [statements](#_Trm00037) 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](#_Trm00020) differ from other [struct types](#_Trm00022) in that they permit certain additional operations:

* Most [simple types](#_Trm00020) permit values to be created by writing *literals* ([§2.4.4](#_Toc00046)). For example, 123 is a [literal](#_Trm00118) of type int and 'a' is a [literal](#_Trm00118) of type char. C# makes no provision for [literal](#_Trm00118)s of [struct types](#_Trm00022) in general, and non-[default value](#_Trm00164)s of other [struct types](#_Trm00022) are ultimately always created through [instance constructor](#_Trm00080)s of those [struct types](#_Trm00022).
* When the [operands](#_Trm00033) 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](#_Grm00068) ([§7.19](#_Toc00346)). [Expressions](#_Trm00032) involving [operator](#_Trm00090)s [defined](#_Trm00121) by other [struct types](#_Trm00022) are not considered to be constant expressions.
* Through const declarations it is possible to declare constants of the [simple types](#_Trm00020) ([§10.4](#_Toc00430)). It is not possible to have constants of other [struct types](#_Trm00022), but a similar effect is provided by static readonly fields.
* Conversions involving [simple types](#_Trm00020) can participate in evaluation of conversion [operator](#_Trm00090)s [defined](#_Trm00121) by other [struct types](#_Trm00022), but a user-[defined](#_Trm00121) conversion [operator](#_Trm00090) can never participate in evaluation of another user-[defined](#_Trm00121) [operator](#_Trm00090) ([§6.4.3](#_Toc00196)).

### Integral [types](#_Trm00011)

C# supports nine integral [types](#_Trm00011): sbyte, byte, short, ushort, int, uint, long, ulong, and char. The integral [types](#_Trm00011) 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 [operator](#_Trm00090)s always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision:

* For the unary + and ~ [operator](#_Trm00090)s, 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](#_Trm00090), 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](#_Trm00090) cannot be applied to [operands](#_Trm00033) of type ulong.
* For the binary +, -, \*, /, %, &, ^, |, ==, !=, >, <, >=, and <= [operator](#_Trm00090)s, the [operands](#_Trm00033) 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](#_Trm00033). The operation is then performed using the precision of type T, and the type of the result is T (or bool for the relational [operator](#_Trm00090)s). It is not permitted for one operand to be of type long and the other to be of type ulong with the binary [operator](#_Trm00090)s.
* For the binary << and >> [operator](#_Trm00090)s, 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](#_Trm00011) in two ways:

* There are no implicit conversions from other [types](#_Trm00011) to the char type. In particular, even though the sbyte, byte, and ushort [types](#_Trm00011) have ranges of values that are fully representable using the char type, implicit conversions from sbyte, byte, or ushort to char do not exist.
* Constants of the char type must be written as [character\_literal](#_Grm00013)s or as [integer\_literal](#_Grm00011)s in combination with a cast to type char. For example, (char)10 is the same as '\x000A'.

The checked and unchecked [operator](#_Trm00090)s and [statements](#_Trm00037) are used to control overflow checking for integral-type arithmetic operations and conversions ([§7.6.12](#_Toc00278)). 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](#_Trm00011)

C# supports two floating point [types](#_Trm00011): float and double. The float and double [types](#_Trm00011) 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 ([§7.8.2](#_Toc00294)).
* 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](#_Trm00033) of a binary [operator](#_Trm00090) 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](#_Trm00033) 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](#_Trm00033) 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 [operator](#_Trm00090)s).
* 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 [operator](#_Trm00090)s).

The floating-point [operator](#_Trm00090)s, including the assignment [operator](#_Trm00090)s, 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](#_Trm00033) 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 decimals with an absolute value less than 1.0m, the value is exact to the 28th decimal place, but no further. For decimals with an absolute value greater than or equal to 1.0m, the value is exact to 28 or 29 digits. Contrary to the float and double data [types](#_Trm00011), 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](#_Trm00033) of a binary [operator](#_Trm00090) 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](#_Trm00121) for each [operator](#_Trm00090)) 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](#_Trm00011). Thus, conversions from the floating-point [types](#_Trm00011) to decimal might produce overflow exceptions, and conversions from decimal to the floating-point [types](#_Trm00011) might cause loss of precision. For these reasons, no implicit conversions exist between the floating-point [types](#_Trm00011) and decimal, and without explicit casts, it is not possible to mix floating-point and decimal [operands](#_Trm00033) 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](#_Trm00011). In particular, the bool type is distinct and separate from the integral [types](#_Trm00011), 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](#_Trm00011)

An enumeration type is a distinct type with named constants. Every enumeration type has an [underlying type](#_Trm00106), 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](#_Trm00106). Values of the enumeration type are not restricted to the values of the named constants. Enumeration [types](#_Trm00011) are [defined](#_Trm00121) through enumeration declarations ([§14.1](#_Toc00552)).

### Nullable [types](#_Trm00011)

A nullable type can represent all values of its ***underlying type*** plus an additional [null value](#_Trm00165). A nullable type is written T?, where T is the [underlying type](#_Trm00106). 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](#_Trm00169) (that is, any type parameter with a struct constraint). The System.Nullable<T> type specifies the value type constraint for T ([§10.1.5](#_Toc00399)), which means that the [underlying type](#_Trm00106) of a nullable type can be any [non-nullable value type](#_Trm00169). The [underlying type](#_Trm00106) of a nullable type cannot be a nullable type or a reference type. For example, int?? and string? are invalid [types](#_Trm00011).

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 un[defined](#_Trm00121) 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](#_Trm00163), 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](#_Trm00118) to T? ([§6.1.5](#_Toc00173)) and from T to T? ([§6.1.4](#_Toc00172)).

## Reference [types](#_Trm00011)

A reference type is a class type, an [interface](#_Trm00102) type, an [array](#_Trm00093) type, or a [delegate type](#_Trm00107).

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](#_Trm00019) and indicates the absence of an [instance](#_Trm00172).

### Class [types](#_Trm00011)

A class type defines a data structure that contains data [members](#_Trm00012) (constants and fields), [function members](#_Trm00079) ([method](#_Trm00056)s, properties, [event](#_Trm00088)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance](#_Trm00172) constructors, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s), and [nested](#_Trm00143) [types](#_Trm00011). Class [types](#_Trm00011) support [inheritance](#_Trm00047), a mechanism whereby [derived classes](#_Trm00049) can extend and specialize [base classes](#_Trm00050). Instances of [class types](#_Trm00024) are created using [object\_creation\_expression](#_Grm00044)s ([§7.6.10.1](#_Toc00271)).

Class [types](#_Trm00011) are described in [§10](#_Toc00388).

Certain pre[defined](#_Trm00121) [class types](#_Trm00024) 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](#_Trm00011). See [§4.2.2](#_Toc00104). |
| System.String | The string type of the C# language. See [§4.2.4](#_Toc00106). |
| System.ValueType | The base class of all value [types](#_Trm00011). See [§4.1.1](#_Toc00092). |
| System.Enum | The base class of all [enum type](#_Trm00105)s. See [§14](#_Toc00551). |
| System.Array | The base class of all [array](#_Trm00093) [types](#_Trm00011). See [§12](#_Toc00518). |
| System.Delegate | The base class of all [delegate type](#_Trm00107)s. See [§15](#_Toc00557). |
| System.Exception | The base class of all exception [types](#_Trm00011). See [§16](#_Toc00562). |

### The [object](#_Trm00173) type

The object class type is the ultimate base class of all other [types](#_Trm00011). Every type in C# directly or indirectly derives from the object class type.

The [keyword](#_Trm00117) object is simply an alias for the pre[defined](#_Trm00121) class System.Object.

### The dynamic type

The dynamic type, like object, can reference any [object](#_Trm00173). When [operator](#_Trm00090)s are applied to expressions of type dynamic, their resolution is deferred until the [program](#_Trm00109) is run. Thus, if the [operator](#_Trm00090) cannot legally be applied to the referenced [object](#_Trm00173), no error is given during compilation. Instead an exception will be thrown when resolution of the [operator](#_Trm00090) fails at run-time.

Its purpose is to allow dynamic binding, which is described in detail in [§7.2.2](#_Toc00209).

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

* Operations on expressions of type dynamic can be dynamically bound ([§7.2.2](#_Toc00209)).
* Type inference ([§7.5.2](#_Toc00227)) 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](#_Trm00011) that are the same when replacing dynamic with object
* Implicit and explicit conversions to and from object also apply to and from dynamic.
* Method [signature](#_Trm00061)s that are the same when replacing dynamic with object are considered the same [signature](#_Trm00061)
* 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 [literal](#_Trm00118)s ([§2.4.4.5](#_Toc00051)).

The [keyword](#_Trm00117) string is simply an alias for the pre[defined](#_Trm00121) class System.String.

### Interface [types](#_Trm00011)

An [interface](#_Trm00102) defines a contract. A class or struct that implements an [interface](#_Trm00102) must adhere to its contract. An [interface](#_Trm00102) may inherit from multiple base [interface](#_Trm00102)s, and a class or struct may implement multiple [interface](#_Trm00102)s.

Interface [types](#_Trm00011) are described in [§13](#_Toc00527).

### Array [types](#_Trm00011)

An [array](#_Trm00093) is a data structure that contains zero or more [variables](#_Trm00031) which are accessed through computed indices. The [variables](#_Trm00031) contained in an [array](#_Trm00093), also called the [elements](#_Trm00094) of the [array](#_Trm00093), are all of the same type, and this type is called the [element type](#_Trm00095) of the [array](#_Trm00093).

Array [types](#_Trm00011) are described in [§12](#_Toc00518).

### Delegate [types](#_Trm00011)

A delegate is a data structure that refers to one or more [method](#_Trm00056)s. For [instance](#_Trm00172) [method](#_Trm00056)s, it also refers to their corresponding [object](#_Trm00173) [instance](#_Trm00172)s.

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](#_Trm00172) [method](#_Trm00056)s. In the latter case, the delegate stores not only a reference to the [method](#_Trm00056)'s [entry point](#_Trm00123), but also a reference to the [object](#_Trm00173) [instance](#_Trm00172) on which to invoke the [method](#_Trm00056).

Delegate [types](#_Trm00011) are described in [§15](#_Toc00557).

## Boxing and un[boxing](#_Trm00029)

The concept of [boxing](#_Trm00029) and un[boxing](#_Trm00029) is central to C#'s type system. It provides a bridge between [value\_type](#_Grm00029)s and [reference\_type](#_Grm00030)s by permitting any value of a [value\_type](#_Grm00029) to be converted to and from type object. Boxing and un[boxing](#_Trm00029) enables a unified view of the type system wherein a value of any type can ultimately be treated as an [object](#_Trm00173).

### Boxing conversions

A [boxing](#_Trm00029) conversion permits a [value\_type](#_Grm00029) to be implicitly converted to a [reference\_type](#_Grm00030). The following [boxing](#_Trm00029) conversions exist:

* From any [value\_type](#_Grm00029) to the type object.
* From any [value\_type](#_Grm00029) to the type System.ValueType.
* From any [non\_nullable\_value\_type](#_Grm00029) to any [interface\_type](#_Grm00030) implemented by the [value\_type](#_Grm00029).
* From any [nullable\_type](#_Grm00029) to any [interface\_type](#_Grm00030) implemented by the [underlying type](#_Trm00106) of the [nullable\_type](#_Grm00029).
* From any [enum\_type](#_Grm00029) to the type System.Enum.
* From any [nullable\_type](#_Grm00029) with an underlying [enum\_type](#_Grm00029) to the type System.Enum.
* Note that an implicit conversion from a type parameter will be executed as a [boxing](#_Trm00029) conversion if at run-time it ends up converting from a value type to a reference type ([§6.1.10](#_Toc00178)).

Boxing a value of a [non\_nullable\_value\_type](#_Grm00029) consists of allocating an [object](#_Trm00173) [instance](#_Trm00172) and copying the [non\_nullable\_value\_type](#_Grm00029) value into that [instance](#_Trm00172).

Boxing a value of a [nullable\_type](#_Grm00029) produces a null reference if it is the null value (HasValue is false), or the result of [unwrapping](#_Trm00170) and [boxing](#_Trm00029) the underlying value otherwise.

The actual process of [boxing](#_Trm00029) a value of a [non\_nullable\_value\_type](#_Grm00029) 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](#_Trm00172) as a value of type object. Thus, the [statements](#_Trm00037)

int i = 123;  
object box = i;

conceptually correspond to

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

A [boxing](#_Trm00029) 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](#_Trm00090) 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](#_Trm00029) conversion implies making a copy of the value being boxed. This is different from a conversion of a [reference\_type](#_Grm00030) to type object, in which the value continues to reference the same [instance](#_Trm00172) 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](#_Trm00037)

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](#_Trm00029) 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](#_Trm00172).

### Un[boxing](#_Trm00029) conversions

An un[boxing](#_Trm00029) conversion permits a [reference\_type](#_Grm00030) to be explicitly converted to a [value\_type](#_Grm00029). The following un[boxing](#_Trm00029) conversions exist:

* From the type object to any [value\_type](#_Grm00029).
* From the type System.ValueType to any [value\_type](#_Grm00029).
* From any [interface\_type](#_Grm00030) to any [non\_nullable\_value\_type](#_Grm00029) that implements the [interface\_type](#_Grm00030).
* From any [interface\_type](#_Grm00030) to any [nullable\_type](#_Grm00029) whose [underlying type](#_Trm00106) implements the [interface\_type](#_Grm00030).
* From the type System.Enum to any [enum\_type](#_Grm00029).
* From the type System.Enum to any [nullable\_type](#_Grm00029) with an underlying [enum\_type](#_Grm00029).
* Note that an explicit conversion to a type parameter will be executed as an un[boxing](#_Trm00029) conversion if at run-time it ends up converting from a reference type to a value type ([§6.2.6](#_Toc00187)).

An un[boxing](#_Trm00029) operation to a [non\_nullable\_value\_type](#_Grm00029) consists of first checking that the [object](#_Trm00173) [instance](#_Trm00172) is a boxed value of the given [non\_nullable\_value\_type](#_Grm00029), and then copying the value out of the [instance](#_Trm00172).

Un[boxing](#_Trm00029) to a [nullable\_type](#_Grm00029) produces the [null value](#_Trm00165) of the [nullable\_type](#_Grm00029) if the source operand is null, or the wrapped result of un[boxing](#_Trm00029) the [object](#_Trm00173) [instance](#_Trm00172) to the [underlying type](#_Trm00106) of the [nullable\_type](#_Grm00029) otherwise.

Referring to the imaginary [boxing](#_Trm00029) class described in the previous section, an un[boxing](#_Trm00029) conversion of an [object](#_Trm00173) box to a [value\_type](#_Grm00029) T consists of executing the expression ((Box<T>)box).value. Thus, the [statements](#_Trm00037)

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

conceptually correspond to

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

For an un[boxing](#_Trm00029) conversion to a given [non\_nullable\_value\_type](#_Grm00029) 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](#_Grm00029). If the source operand is null, a System.NullReferenceException is thrown. If the source operand is a reference to an incompatible [object](#_Trm00173), a System.InvalidCastException is thrown.

For an un[boxing](#_Trm00029) conversion to a given [nullable\_type](#_Grm00029) 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](#_Grm00029) of the [nullable\_type](#_Grm00029). If the source operand is a reference to an incompatible [object](#_Trm00173), a System.InvalidCastException is thrown.

## Constructed [types](#_Trm00011)

A generic type declaration, by itself, denotes an ***unbound generic type*** that is used as a "blueprint" to form many different [types](#_Trm00011), by way of applying ***type arguments***. The type [arguments](#_Trm00062) 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](#_Trm00178) can be used in most places in the language in which a type name can appear. An [unbound generic type](#_Trm00176) can only be used within a [typeof\_expression](#_Grm00050) ([§7.6.11](#_Toc00277)).

Constructed [types](#_Trm00011) can also be used in expressions as simple names ([§7.6.2](#_Toc00254)) or when accessing a member ([§7.6.4](#_Toc00257)).

When a [namespace\_or\_type\_name](#_Grm00027) is evaluated, only [generic types](#_Trm00158) with the correct number of type [parameters](#_Trm00059) are considered. Thus, it is possible to use the same identifier to identify different [types](#_Trm00011), as long as the [types](#_Trm00011) have different numbers of type [parameters](#_Trm00059). This is useful when mixing generic and non-generic classes in the same [program](#_Trm00109):

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](#_Grm00027) might identify a [constructed type](#_Trm00178) even though it doesn't specify type [parameters](#_Trm00059) directly. This can occur where a type is [nested](#_Trm00143) within a generic class declaration, and the [instance](#_Trm00172) type of the containing declaration is implicitly used for name lookup ([§10.3.8.6](#_Toc00424)):

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

In unsafe code, a [constructed type](#_Trm00178) cannot be used as an [unmanaged\_type](#_Grm00150) ([§18.2](#_Toc00592)).

### Type [arguments](#_Trm00062)

Each argument in a type argument list is simply a [type](#_Grm00028).

type\_argument\_list:  
 | '<' type\_arguments '>'  
 ;  
  
type\_arguments:  
 | type\_argument ( ',' type\_argument )\*  
 ;  
  
type\_argument:  
 | type  
 ;

In unsafe code ([§18](#_Toc00590)), a [type\_argument](#_Grm00031) may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter ([§10.1.5](#_Toc00399)).

### Open and closed [types](#_Trm00011)

All [types](#_Trm00011) can be classified as either ***open types*** or ***closed types***. An open type is a type that involves type [parameters](#_Trm00059). More specifically:

* A type parameter defines an open type.
* An [array](#_Trm00093) type is an open type if and only if its [element type](#_Trm00095) is an open type.
* A [constructed type](#_Trm00178) is an open type if and only if one or more of its type [arguments](#_Trm00062) is an open type. A constructed [nested](#_Trm00143) type is an open type if and only if one or more of its type [arguments](#_Trm00062) or the type [arguments](#_Trm00062) 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](#_Trm00178) that was created by applying type [arguments](#_Trm00062) to the generic declaration. Each type parameter within the generic type is bound to a particular [run-time type](#_Trm00073). The run-time processing of all [statements](#_Trm00037) and expressions always occurs with [closed types](#_Trm00180), and [open types](#_Trm00179) occur only during compile-time processing.

Each closed [constructed type](#_Trm00178) has its own set of static [variables](#_Trm00031), which are not shared with any other closed [constructed type](#_Trm00178)s. Since an open type does not exist at run-time, there are no static [variables](#_Trm00031) associated with an open type. Two closed [constructed type](#_Trm00178)s are the same type if they are constructed from the same [unbound generic type](#_Trm00176), and their corresponding type [arguments](#_Trm00062) are the same type.

### Bound and unbound [types](#_Trm00011)

The term ***unbound type*** refers to a non-generic type or an [unbound generic type](#_Trm00176). The term ***bound type*** refers to a non-generic type or a [constructed type](#_Trm00178).

An un[bound type](#_Trm00182) refers to the entity declared by a type declaration. An [unbound generic type](#_Trm00176) 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](#_Trm00176) can be referenced is the typeof expression ([§7.6.11](#_Toc00277)).

### Satisfying constraints

Whenever a [constructed type](#_Trm00178) or generic [method](#_Trm00056) is referenced, the supplied type [arguments](#_Trm00062) are checked against the type parameter constraints declared on the generic type or [method](#_Trm00056) ([§10.1.5](#_Toc00399)). 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](#_Trm00102) type, or a type parameter, let C represent that constraint with the supplied type [arguments](#_Trm00062) substituted for any type [parameters](#_Trm00059) 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 ([§6.1.1](#_Toc00169))
  + An implicit reference conversion ([§6.1.6](#_Toc00174))
  + A [boxing](#_Trm00029) conversion ([§6.1.7](#_Toc00175)), provided that type A is a [non-nullable value type](#_Trm00169).
  + An implicit reference, [boxing](#_Trm00029) 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](#_Trm00102) type, class type, [delegate type](#_Trm00107) or [array](#_Trm00093) type. Note that System.ValueType and System.Enum are [reference types](#_Trm00019) that satisfy this constraint.
  + A is a type parameter that is known to be a reference type ([§10.1.5](#_Toc00399)).
* 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](#_Trm00105), but not a nullable type. Note that System.ValueType and System.Enum are [reference types](#_Trm00019) that do not satisfy this constraint.
  + A is a type parameter having the value type constraint ([§10.1.5](#_Toc00399)).
* 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](#_Trm00011) have a public [default constructor](#_Trm00163) ([§4.1.2](#_Toc00093)).
  + A is a type parameter having the constructor constraint ([§10.1.5](#_Toc00399)).
  + A is a type parameter having the value type constraint ([§10.1.5](#_Toc00399)).
  + A is a class that is not abstract and contains an explicitly declared public constructor with no [parameters](#_Trm00059).
  + A is not abstract and has a [default constructor](#_Trm00163) ([§10.11.4](#_Toc00478)).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type [arguments](#_Trm00062).

Since type [parameters](#_Trm00059) are not [inherited](#_Trm00136), constraints are never [inherited](#_Trm00136) 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](#_Trm00059)

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](#_Trm00062), type [parameters](#_Trm00059) have slightly different operations and restrictions than other [types](#_Trm00011). These include:

* A type parameter cannot be used directly to declare a base class ([§10.2.4](#_Toc00405)) or [interface](#_Trm00102) ([§13.1.3](#_Toc00531)).
* The rules for member lookup on type [parameters](#_Trm00059) depend on the constraints, if any, applied to the type parameter. They are detailed in [§7.4](#_Toc00221).
* The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in [§6.1.10](#_Toc00178) and [§6.2.6](#_Toc00187).
* The [literal](#_Trm00118) null cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type ([§6.1.10](#_Toc00178)). However, a default expression ([§7.6.13](#_Toc00279)) can be used instead. In addition, a value with a type given by a type parameter can be compared with null using == and != ([§7.10.6](#_Toc00305)) unless the type parameter has the value type constraint.
* A new expression ([§7.6.10.1](#_Toc00271)) can only be used with a type parameter if the type parameter is constrained by a [constructor\_constraint](#_Grm00111) or the value type constraint ([§10.1.5](#_Toc00399)).
* A type parameter cannot be used anywhere within an attribute.
* A type parameter cannot be used in a member access ([§7.6.4](#_Toc00257)) or type name ([§3.8](#_Toc00086)) to identify a static member or a [nested](#_Trm00143) type.
* In unsafe code, a type parameter cannot be used as an [unmanaged\_type](#_Grm00150) ([§18.2](#_Toc00592)).

As a type, type [parameters](#_Trm00059) are purely a compile-time construct. At run-time, each type parameter is bound to a [run-time type](#_Trm00073) 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](#_Trm00178) ([§4.4.2](#_Toc00115)). The run-time execution of all [statements](#_Trm00037) and expressions involving type [parameters](#_Trm00059) uses the actual type that was supplied as the type argument for that parameter.

## Expression tree [types](#_Trm00011)

***Expression trees*** permit lambda expressions to be represented as data structures instead of executable code. [Expression trees](#_Trm00183) are values of ***expression tree types*** of the form System.Linq.Expressions.Expression<D>, where D is any [delegate type](#_Trm00107). For the remainder of this specification we will refer to these [types](#_Trm00011) using the shorthand Expression<D>.

If a conversion exists from a lambda expression to a [delegate type](#_Trm00107) D, a conversion also exists to the expression tree type Expression<D>. Whereas the conversion of a lambda expression to a [delegate type](#_Trm00107) generates a delegate that [references](#_Trm00160) executable code for the lambda expression, conversion to an expression tree type creates an expression tree representation of the lambda expression.

[Expression trees](#_Trm00183) are efficient in-memory data representations of lambda expressionsand make the structure of the lambda expressiontransparent and explicit.

Just like a [delegate type](#_Trm00107) D, Expression<D> is said to have parameter and [return type](#_Trm00060)s, which are the same as those of D.

The following example represents a lambda expressionboth 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](#_Trm00160) a [method](#_Trm00056) that returns x + 1, and the expression tree exp [references](#_Trm00160) 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 expressionis converted to an expression tree type, are both outside the [scope](#_Trm00148) of this specification.

Two things are important to make explicit:

* Not all lambda expressions can be converted to expression trees. For [instance](#_Trm00172), 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 [§6.5](#_Toc00199).
* Expression<D> offers an [instance](#_Trm00172) [method](#_Trm00056) 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](#_Trm00037) 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

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a [type-safe](#_Trm00006) language, and the C# compiler guarantees that values stored in [variables](#_Trm00031) are always of the appropriate type. The value of a variable can be changed through assignment or through use of the ++ and -- [operator](#_Trm00090)s.

A variable must be ***definitely assigned*** ([§5.3](#_Toc00132)) before its value can be obtained.

As described in the following sections, [variables](#_Trm00031) are either ***initially assigned*** or ***initially unassigned***. An [initially assigned](#_Trm00186) variable has a well-[defined](#_Trm00121) initial value and is always considered [definitely assigned](#_Trm00068). An [initially unassigned](#_Trm00187) variable has no initial value. For an [initially unassigned](#_Trm00187) variable to be considered [definitely assigned](#_Trm00068) 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](#_Trm00031): static [variables](#_Trm00031), [instance](#_Trm00172) [variables](#_Trm00031), [array](#_Trm00093) [elements](#_Trm00094), value [parameters](#_Trm00059), reference [parameters](#_Trm00059), [output parameter](#_Trm00065)s, and [local variables](#_Trm00067). 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](#_Trm00172) variable, v[0] is an [array](#_Trm00093) element, a is a [value parameter](#_Trm00063), b is a [reference parameter](#_Trm00064), c is an [output parameter](#_Trm00065), and i is a local variable.

### Static [variables](#_Trm00031)

A field declared with the static modifier is called a ***static variable***. A [static variable](#_Trm00188) comes into existence before execution of the [static constructor](#_Trm00081) ([§10.12](#_Toc00481)) for its containing type, and ceases to exist when the associated [application](#_Trm00124) domain ceases to exist.

The initial value of a [static variable](#_Trm00188) is the [default value](#_Trm00164) ([§5.2](#_Toc00131)) of the variable's type.

For purposes of definite assignment checking, a [static variable](#_Trm00188) is considered [initially assigned](#_Trm00186).

### Instance [variables](#_Trm00031)

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

#### Instance [variables](#_Trm00031) in classes

An [instance](#_Trm00172) variable of a class comes into existence when a new [instance](#_Trm00172) of that class is created, and ceases to exist when there are no [references](#_Trm00160) to that [instance](#_Trm00172) and the [instance](#_Trm00172)'s [destructor](#_Trm00091) (if any) has executed.

The initial value of an [instance](#_Trm00172) variable of a class is the [default value](#_Trm00164) ([§5.2](#_Toc00131)) of the variable's type.

For the purpose of definite assignment checking, an [instance](#_Trm00172) variable of a class is considered [initially assigned](#_Trm00186).

#### Instance [variables](#_Trm00031) in [structs](#_Trm00092)

An [instance](#_Trm00172) 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](#_Trm00172) [variables](#_Trm00031) of the struct.

The initial assignment state of an [instance](#_Trm00172) 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](#_Trm00186), so too are its [instance](#_Trm00172) [variables](#_Trm00031), and when a struct variable is considered [initially unassigned](#_Trm00187), its [instance](#_Trm00172) [variables](#_Trm00031) are likewise unassigned.

### Array [elements](#_Trm00094)

The [elements](#_Trm00094) of an [array](#_Trm00093) come into existence when an [array](#_Trm00093) [instance](#_Trm00172) is created, and cease to exist when there are no [references](#_Trm00160) to that [array](#_Trm00093) [instance](#_Trm00172).

The initial value of each of the [elements](#_Trm00094) of an [array](#_Trm00093) is the [default value](#_Trm00164) ([§5.2](#_Toc00131)) of the type of the [array](#_Trm00093) [elements](#_Trm00094).

For the purpose of definite assignment checking, an [array](#_Trm00093) element is considered [initially assigned](#_Trm00186).

### Value [parameters](#_Trm00059)

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

A [value parameter](#_Trm00063) comes into existence upon invocation of the function member ([method](#_Trm00056), [instance](#_Trm00172) constructor, accessor, or [operator](#_Trm00090)) or anonymous function to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A [value parameter](#_Trm00063) normally ceases to exist upon return of the function member or anonymous function. However, if the [value parameter](#_Trm00063) is captured by an anonymous function ([§7.15](#_Toc00321)), 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](#_Trm00063) is considered [initially assigned](#_Trm00186).

### Reference [parameters](#_Trm00059)

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

A [reference parameter](#_Trm00064) does not create a new storage location. Instead, a [reference parameter](#_Trm00064) 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](#_Trm00064) is always the same as the underlying variable.

The following definite assignment rules apply to reference [parameters](#_Trm00059). Note the different rules for [output parameter](#_Trm00065)s described in [§5.1.6](#_Toc00129).

* A variable must be [definitely assigned](#_Trm00068) ([§5.3](#_Toc00132)) before it can be passed as a [reference parameter](#_Trm00064) in a function member or delegate invocation.
* Within a function member or anonymous function, a [reference parameter](#_Trm00064) is considered [initially assigned](#_Trm00186).

Within an [instance](#_Trm00172) [method](#_Trm00056) or [instance](#_Trm00172) accessor of a struct type, the this [keyword](#_Trm00117) behaves exactly as a [reference parameter](#_Trm00064) of the struct type ([§7.6.7](#_Toc00267)).

### Output [parameters](#_Trm00059)

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

An [output parameter](#_Trm00065) does not create a new storage location. Instead, an [output parameter](#_Trm00065) 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](#_Trm00065) is always the same as the underlying variable.

The following definite assignment rules apply to [output parameter](#_Trm00065)s. Note the different rules for reference [parameters](#_Trm00059) described in [§5.1.5](#_Toc00128).

* A variable need not be [definitely assigned](#_Trm00068) before it can be passed as an [output parameter](#_Trm00065) 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](#_Trm00065) is considered assigned in that execution path.
* Within a function member or anonymous function, an [output parameter](#_Trm00065) is considered [initially unassigned](#_Trm00187).
* Every [output parameter](#_Trm00065) of a function member or anonymous function must be [definitely assigned](#_Trm00068) ([§5.3](#_Toc00132)) before the function member or anonymous function returns normally.

Within an [instance](#_Trm00172) constructor of a struct type, the this [keyword](#_Trm00117) behaves exactly as an [output parameter](#_Trm00065) of the struct type ([§7.6.7](#_Toc00267)).

### Local [variables](#_Trm00031)

A ***local variable*** is declared by a [local\_variable\_declaration](#_Grm00076), which may occur in a [block](#_Grm00071), a [for\_statement](#_Grm00085), a [switch\_statement](#_Grm00081) or a [using\_statement](#_Grm00096); or by a [foreach\_statement](#_Grm00086) or a [specific\_catch\_clause](#_Grm00093) for a [try\_statement](#_Grm00093).

The lifetime of a [local variable](#_Trm00193) is the portion of [program](#_Trm00109) execution during which storage is guaranteed to be reserved for it. This lifetime extends at least from entry into the [block](#_Grm00071), [for\_statement](#_Grm00085), [switch\_statement](#_Grm00081), [using\_statement](#_Grm00096), [foreach\_statement](#_Grm00086), or [specific\_catch\_clause](#_Grm00093) with which it is associated, until execution of that [block](#_Grm00071), [for\_statement](#_Grm00085), [switch\_statement](#_Grm00081), [using\_statement](#_Grm00096), [foreach\_statement](#_Grm00086), or [specific\_catch\_clause](#_Grm00093) ends in any way. (Entering an enclosed [block](#_Grm00071) or calling a [method](#_Trm00056) suspends, but does not end, execution of the current [block](#_Grm00071), [for\_statement](#_Grm00085), [switch\_statement](#_Grm00081), [using\_statement](#_Grm00096), [foreach\_statement](#_Grm00086), or [specific\_catch\_clause](#_Grm00093).) If the [local variable](#_Trm00193) is captured by an anonymous function ([§7.15.5.1](#_Toc00327)), its lifetime extends at least until the delegate or expression tree created from the anonymous function, along with any other [object](#_Trm00173)s that come to reference the captured variable, are eligible for garbage collection.

If the parent [block](#_Grm00071), [for\_statement](#_Grm00085), [switch\_statement](#_Grm00081), [using\_statement](#_Grm00096), [foreach\_statement](#_Grm00086), or [specific\_catch\_clause](#_Grm00093) is entered recursively, a new [instance](#_Trm00172) of the [local variable](#_Trm00193) is created each time, and its [local\_variable\_initializer](#_Grm00076), if any, is evaluated each time.

A [local variable](#_Trm00193) introduced by a [local\_variable\_declaration](#_Grm00076) is not automatically initialized and thus has no [default value](#_Trm00164). For the purpose of definite assignment checking, a [local variable](#_Trm00193) introduced by a [local\_variable\_declaration](#_Grm00076) is considered [initially unassigned](#_Trm00187). A [local\_variable\_declaration](#_Grm00076) may include a [local\_variable\_initializer](#_Grm00076), in which case the variable is considered [definitely assigned](#_Trm00068) only after the initializing expression ([§5.3.3.4](#_Toc00139)).

Within the [scope](#_Trm00148) of a [local variable](#_Trm00193)introduced by a [local\_variable\_declaration](#_Grm00076), it is a compile-time error to refer to that [local variable](#_Trm00193) in a textual position that precedes its [local\_variable\_declarator](#_Grm00076). If the [local variable](#_Trm00193) declaration is implicit ([§8.5.1](#_Toc00355)), it is also an error to refer to the variable within its [local\_variable\_declarator](#_Grm00076).

A [local variable](#_Trm00193) introduced by a [foreach\_statement](#_Grm00086) or a [specific\_catch\_clause](#_Grm00093) is considered [definitely assigned](#_Trm00068) in its entire [scope](#_Trm00148).

The actual lifetime of a [local variable](#_Trm00193) is implementation-dependent. For example, a compiler might statically determine that a [local variable](#_Trm00193) in a [block](#_Trm00038) is only used for a small portion of that [block](#_Trm00038). Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing [block](#_Trm00038).

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable ([§3.9](#_Toc00088)).

## Default values

The following categories of [variables](#_Trm00031) are automatically initialized to their [default value](#_Trm00164)s:

* Static [variables](#_Trm00031).
* Instance [variables](#_Trm00031) of class [instance](#_Trm00172)s.
* Array [elements](#_Trm00094).

The [default value](#_Trm00164) of a variable depends on the type of the variable and is determined as follows:

* For a variable of a [value\_type](#_Grm00029), the [default value](#_Trm00164) is the same as the value computed by the [value\_type](#_Grm00029)'s [default constructor](#_Trm00163) ([§4.1.2](#_Toc00093)).
* For a variable of a [reference\_type](#_Grm00030), the [default value](#_Trm00164) is null.

Initialization to [default value](#_Trm00164)s is typically done by having the memory manager or [garbage collector](#_Trm00154) 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 ([§5.3.3](#_Toc00135)), 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](#_Trm00186) variable ([§5.3.1](#_Toc00133)) is always considered [definitely assigned](#_Trm00068).
* An [initially unassigned](#_Trm00187) variable ([§5.3.2](#_Toc00134)) is considered [definitely assigned](#_Trm00068) at a given location if all possible execution paths leading to that location contain at least one of the following:
  + A simple assignment ([§7.17.1](#_Toc00342)) in which the variable is the left operand.
  + An invocation expression ([§7.6.5](#_Toc00260)) or [object](#_Trm00173) creation expression ([§7.6.10.1](#_Toc00271)) that passes the variable as an [output parameter](#_Trm00065).
  + For a [local variable](#_Trm00193), a [local variable](#_Trm00193) declaration ([§8.5.1](#_Toc00355)) that includes a variable initializer.

The formal specification underlying the above informal rules is described in [§5.3.1](#_Toc00133), [§5.3.2](#_Toc00134), and [§5.3.3](#_Toc00135).

The definite assignment states of [instance](#_Trm00172) [variables](#_Trm00031) of a [struct\_type](#_Grm00029) variable are tracked individually as well as collectively. In additional to the rules above, the following rules apply to [struct\_type](#_Grm00029) [variables](#_Trm00031) and their [instance](#_Trm00172) [variables](#_Trm00031):

* An [instance](#_Trm00172) variable is considered [definitely assigned](#_Trm00068) if its containing [struct\_type](#_Grm00029) variable is considered [definitely assigned](#_Trm00068).
* A [struct\_type](#_Grm00029) variable is considered [definitely assigned](#_Trm00068) if each of its [instance](#_Trm00172) [variables](#_Trm00031) is considered [definitely assigned](#_Trm00068).

Definite assignment is a requirement in the following contexts:

* A variable must be [definitely assigned](#_Trm00068) at each location where its value is obtained. This ensures that un[defined](#_Trm00121) 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](#_Trm00065), or
  + the variable is a [struct\_type](#_Grm00029) variable and occurs as the left operand of a member access.
* A variable must be [definitely assigned](#_Trm00068) at each location where it is passed as a [reference parameter](#_Trm00064). This ensures that the function member being invoked can consider the [reference parameter](#_Trm00064) [initially assigned](#_Trm00186).
* All [output parameter](#_Trm00065)s of a function member must be [definitely assigned](#_Trm00068) 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](#_Trm00079) do not return un[defined](#_Trm00121) values in [output parameter](#_Trm00065)s, thus enabling the compiler to consider a function member invocation that takes a variable as an [output parameter](#_Trm00065) equivalent to an assignment to the variable.
* The this variable of a [struct\_type](#_Grm00029) [instance](#_Trm00172) constructor must be [definitely assigned](#_Trm00068) at each location where that [instance](#_Trm00172) constructor returns.

### Initially assigned [variables](#_Trm00031)

The following categories of [variables](#_Trm00031) are classified as [initially assigned](#_Trm00186):

* Static [variables](#_Trm00031).
* Instance [variables](#_Trm00031) of class [instance](#_Trm00172)s.
* Instance [variables](#_Trm00031) of [initially assigned](#_Trm00186) struct [variables](#_Trm00031).
* Array [elements](#_Trm00094).
* Value [parameters](#_Trm00059).
* Reference [parameters](#_Trm00059).
* Variables declared in a catch clause or a foreach statement.

### Initially unassigned [variables](#_Trm00031)

The following categories of [variables](#_Trm00031) are classified as [initially unassigned](#_Trm00187):

* Instance [variables](#_Trm00031) of [initially unassigned](#_Trm00187) struct [variables](#_Trm00031).
* Output [parameters](#_Trm00059), including the this variable of struct [instance](#_Trm00172) constructors.
* Local [variables](#_Trm00031), 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](#_Trm00068), 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](#_Trm00187) [variables](#_Trm00031). For each [initially unassigned](#_Trm00187) 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 ([§8.1](#_Toc00349)) 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](#_Trm00195) 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](#_Trm00068). 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](#_Trm00068) 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](#_Trm00068) 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](#_Trm00068) 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](#_Trm00037)

* *v* is not [definitely assigned](#_Trm00068) at the beginning of a function member body.
* *v* is [definitely assigned](#_Trm00068) at the beginning of any unreachable statement.
* The [definite assignment state](#_Trm00195) of *v* at the beginning of any other statement is determined by checking the [definite assignment state](#_Trm00195) of *v* on all control flow transfers that target the beginning of that statement. If (and only if) *v* is [definitely assigned](#_Trm00068) on all such control flow transfers, then *v* is [definitely assigned](#_Trm00068) at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability ([§8.1](#_Toc00349)).
* The [definite assignment state](#_Trm00195) of *v* at the end point of a [block](#_Trm00038), checked, unchecked, if, while, do, for, foreach, lock, using, or switch statement is determined by checking the [definite assignment state](#_Trm00195) of *v* on all control flow transfers that target the end point of that statement. If *v* is [definitely assigned](#_Trm00068) on all such control flow transfers, then *v* is [definitely assigned](#_Trm00068) at the end point of the statement. Otherwise; *v* is not [definitely assigned](#_Trm00068) 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 ([§8.1](#_Toc00349)).

#### Block [statements](#_Trm00037), checked, and unchecked [statements](#_Trm00037)

The [definite assignment state](#_Trm00195) of *v* on the control transfer to the first statement of the statement list in the [block](#_Trm00038) (or to the end point of the [block](#_Trm00038), if the statement list is empty) is the same as the [definite assignment state](#_Trm00195)ment of *v* before the [block](#_Trm00038), checked, or unchecked statement.

#### [Expression statements](#_Trm00040)

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

* *v* has the same [definite assignment state](#_Trm00195) at the beginning of *expr* as at the beginning of *stmt*.
* If *v* if [definitely assigned](#_Trm00068) at the end of *expr*, it is [definitely assigned](#_Trm00068) at the end point of *stmt*; otherwise; it is not [definitely assigned](#_Trm00068) at the end point of *stmt*.

#### [Declaration statements](#_Trm00039)

* If *stmt* is a declaration statement without initializers, then *v* has the same [definite assignment state](#_Trm00195) 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](#_Trm00195) 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](#_Trm00037)

For an if statement *stmt* of the form:

if ( expr ) then\_stmt else else\_stmt

* *v* has the same [definite assignment state](#_Trm00195) at the beginning of *expr* as at the beginning of *stmt*.
* If *v* is [definitely assigned](#_Trm00068) at the end of *expr*, then it is [definitely assigned](#_Trm00068) 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](#_Trm00068) after true expression" at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to *then\_stmt*, and not [definitely assigned](#_Trm00068) 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](#_Trm00068) after false expression" at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to *else\_stmt*, and not [definitely assigned](#_Trm00068) on the control flow transfer to *then\_stmt*. It is [definitely assigned](#_Trm00068) at the end-point of *stmt* if and only if it is [definitely assigned](#_Trm00068) at the end-point of *then\_stmt*.
* Otherwise, *v* is considered not [definitely assigned](#_Trm00068) 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](#_Trm00037)

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

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* on the control flow transfer to a reachable switch [block](#_Trm00038) statement list is the same as the [definite assignment state](#_Trm00195) of *v* at the end of *expr*.

#### While [statements](#_Trm00037)

For a while statement *stmt* of the form:

while ( expr ) while\_body

* *v* has the same [definite assignment state](#_Trm00195) at the beginning of *expr* as at the beginning of *stmt*.
* If *v* is [definitely assigned](#_Trm00068) at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to *while\_body* and to the end point of *stmt*.
* If *v* has the state "[definitely assigned](#_Trm00068) after true expression" at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to *while\_body*, but not [definitely assigned](#_Trm00068) at the end-point of *stmt*.
* If *v* has the state "[definitely assigned](#_Trm00068) after false expression" at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to the end point of *stmt*, but not [definitely assigned](#_Trm00068) on the control flow transfer to *while\_body*.

#### Do [statements](#_Trm00037)

For a do statement *stmt* of the form:

do do\_body while ( expr ) ;

* *v* has the same [definite assignment state](#_Trm00195) 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](#_Trm00195) at the beginning of *expr* as at the end point of *do\_body*.
* If *v* is [definitely assigned](#_Trm00068) at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to the end point of *stmt*.
* If *v* has the state "[definitely assigned](#_Trm00068) after false expression" at the end of *expr*, then it is [definitely assigned](#_Trm00068) on the control flow transfer to the end point of *stmt*.

#### For [statements](#_Trm00037)

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](#_Grm00085) is omitted from the for statement, then evaluation of definite assignment proceeds as if [for\_condition](#_Grm00085) were replaced with true in the above expansion.

#### Break, continue, and goto [statements](#_Trm00037)

The [definite assignment state](#_Trm00195) of *v* on the control flow transfer caused by a break, continue, or goto statement is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of the statement.

#### Throw [statements](#_Trm00037)

For a statement *stmt* of the form

throw expr ;

The [definite assignment state](#_Trm00195) of *v* at the beginning of *expr* is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of *stmt*.

#### Return [statements](#_Trm00037)

For a statement *stmt* of the form

return expr ;

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *expr* is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of *stmt*.
* If *v* is an [output parameter](#_Trm00065), then it must be [definitely assigned](#_Trm00068) either:
  + after *expr*
  + or at the end of the finally [block](#_Trm00038) 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](#_Trm00065), then it must be [definitely assigned](#_Trm00068) either:
  + before *stmt*
  + or at the end of the finally [block](#_Trm00038) of a try-finally or try-catch-finally that encloses the return statement.

#### Try-catch [statements](#_Trm00037)

For a statement *stmt* of the form:

try try\_block  
catch(...) catch\_block\_1  
...  
catch(...) catch\_block\_n

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *try\_block* is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* at the beginning of *catch\_block\_i* (for any *i*) is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* at the end-point of *stmt* is [definitely assigned](#_Trm00068) if (and only if) *v* is [definitely assigned](#_Trm00068) at the end-point of *try\_block* and every *catch\_block\_i* (for every *i* from 1 to *n*).

#### Try-finally [statements](#_Trm00037)

For a try statement *stmt* of the form:

try try\_block finally finally\_block

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *try\_block* is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* at the beginning of *finally\_block* is the same as the [definite assignment state](#_Trm00195) of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* at the end-point of *stmt* is [definitely assigned](#_Trm00068) if (and only if) at least one of the following is true:
  + *v* is [definitely assigned](#_Trm00068) at the end-point of *try\_block*
  + *v* is [definitely assigned](#_Trm00068) 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](#_Trm00068) on that control flow transfer if *v* is [definitely assigned](#_Trm00068) at the end-point of *finally\_block*. (This is not an only if—if *v* is [definitely assigned](#_Trm00068) for another reason on this control flow transfer, then it is still considered [definitely assigned](#_Trm00068).)

#### Try-catch-finally [statements](#_Trm00037)

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 [block](#_Trm00038)s of a try statement ([§8.10](#_Toc00372)) 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](#_Trm00037)

For a foreach statement *stmt* of the form:

foreach ( type identifier in expr ) embedded\_statement

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* on the control flow transfer to [embedded\_statement](#_Grm00070) or to the end point of *stmt* is the same as the state of *v* at the end of *expr*.

#### Using [statements](#_Trm00037)

For a using statement *stmt* of the form:

using ( resource\_acquisition ) embedded\_statement

* The [definite assignment state](#_Trm00195) of *v* at the beginning of [resource\_acquisition](#_Grm00096) is the same as the state of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* on the control flow transfer to [embedded\_statement](#_Grm00070) is the same as the state of *v* at the end of [resource\_acquisition](#_Grm00096).

#### Lock [statements](#_Trm00037)

For a lock statement *stmt* of the form:

lock ( expr ) embedded\_statement

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) of *v* on the control flow transfer to [embedded\_statement](#_Grm00070) is the same as the state of *v* at the end of *expr*.

#### Yield [statements](#_Trm00037)

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

yield return expr ;

* The [definite assignment state](#_Trm00195) of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
* The [definite assignment state](#_Trm00195) 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](#_Trm00195).

#### General rules for simple expressions

The following rule applies to these kinds of expressions: [literal](#_Trm00118)s ([§7.6.1](#_Toc00253)), simple names ([§7.6.2](#_Toc00254)), member access expressions ([§7.6.4](#_Toc00257)), non-indexed base access expressions ([§7.6.8](#_Toc00268)), typeof expressions ([§7.6.11](#_Toc00277)), and [default value](#_Trm00164) expressions ([§7.6.13](#_Toc00279)).

* The [definite assignment state](#_Trm00195) of *v* at the end of such an expression is the same as the [definite assignment state](#_Trm00195) 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 ([§7.6.3](#_Toc00256)), element access expressions ([§7.6.6](#_Toc00264)), base access expressions with indexing ([§7.6.8](#_Toc00268)), increment and decrement expressions ([§7.6.9](#_Toc00269), [§7.7.5](#_Toc00286)), cast expressions ([§7.7.6](#_Toc00287)), unary +, -, ~, \* expressions, binary +, -, \*, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ expressions ([§7.8](#_Toc00292), [§7.9](#_Toc00298), [§7.10](#_Toc00299), [§7.11](#_Toc00311)), compound assignment expressions ([§7.17.2](#_Toc00343)), checked and unchecked expressions ([§7.6.12](#_Toc00278)), plus [array](#_Trm00093) and delegate creation expressions ([§7.6.10](#_Toc00270)).

Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed order. For example, the binary % [operator](#_Trm00090) evaluates the left hand side of the [operator](#_Trm00090), 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](#_Trm00195) of *v* at the beginning of *e1* is the same as the [definite assignment state](#_Trm00195) at the beginning of *expr*.
* The [definite assignment state](#_Trm00195) of *v* at the beginning of *ei* (*i* greater than one) is the same as the [definite assignment state](#_Trm00195) at the end of the previous sub-expression.
* The [definite assignment state](#_Trm00195) of *v* at the end of *expr* is the same as the [definite assignment state](#_Trm00195) at the end of *eN*

#### Invocation expressions and [object](#_Trm00173) creation expressions

For an invocation expression *expr* of the form:

primary\_expression ( arg1 , arg2 , ... , argN )

or an [object](#_Trm00173) creation expression of the form:

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

* For an invocation expression, the [definite assignment state](#_Trm00195) of *v* before [primary\_expression](#_Grm00035) is the same as the state of *v* before *expr*.
* For an invocation expression, the [definite assignment state](#_Trm00195) of *v* before *arg1* is the same as the state of *v* after [primary\_expression](#_Grm00035).
* For an [object](#_Trm00173) creation expression, the [definite assignment state](#_Trm00195) of *v* before *arg1* is the same as the state of *v* before *expr*.
* For each argument *argi*, the [definite assignment state](#_Trm00195) 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](#_Trm00195) 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](#_Trm00062), then the state of *v* after *expr* is [definitely assigned](#_Trm00068). Otherwise; the state of *v* after *expr* is the same as the state of *v* after *argN*.
* For [array](#_Trm00093) initializers ([§7.6.10.4](#_Toc00274)), [object](#_Trm00173) initializers ([§7.6.10.2](#_Toc00272)), collection initializers ([§7.6.10.3](#_Toc00273)) and anonymous [object](#_Trm00173) initializers ([§7.6.10.6](#_Toc00276)), the [definite assignment state](#_Trm00195) is determined by the expansion that these con[structs](#_Trm00092) are [defined](#_Trm00121) in terms of.

#### Simple assignment expressions

For an expression *expr* of the form w = expr\_rhs:

* The [definite assignment state](#_Trm00195) of *v* before *expr\_rhs* is the same as the [definite assignment state](#_Trm00195) of *v* before *expr*.
* If *w* is the same variable as *v*, then the [definite assignment state](#_Trm00195) of *v* after *expr* is [definitely assigned](#_Trm00068). Otherwise, the [definite assignment state](#_Trm00195) of *v* after *expr* is the same as the [definite assignment state](#_Trm00195) of *v* after *expr\_rhs*.

#### && (conditional AND) expressions

For an expression *expr* of the form expr\_first && expr\_second:

* The [definite assignment state](#_Trm00195) of *v* before *expr\_first* is the same as the [definite assignment state](#_Trm00195) of *v* before *expr*.
* The [definite assignment state](#_Trm00195) of *v* before *expr\_second* is [definitely assigned](#_Trm00068) if the state of *v* after *expr\_first* is either [definitely assigned](#_Trm00068) or "[definitely assigned](#_Trm00068) after true expression". Otherwise, it is not [definitely assigned](#_Trm00068).
* The [definite assignment state](#_Trm00195) of *v* after *expr* is determined by:
  + If *expr\_first* is a constant expression with the value false, then the [definite assignment state](#_Trm00195) of *v* after *expr* is the same as the [definite assignment state](#_Trm00195) of *v* after *expr\_first*.
  + Otherwise, if the state of *v* after *expr\_first* is [definitely assigned](#_Trm00068), then the state of *v* after *expr* is [definitely assigned](#_Trm00068).
  + Otherwise, if the state of *v* after *expr\_second* is [definitely assigned](#_Trm00068), and the state of *v* after *expr\_first* is "[definitely assigned](#_Trm00068) after false expression", then the state of *v* after *expr* is [definitely assigned](#_Trm00068).
  + Otherwise, if the state of *v* after *expr\_second* is [definitely assigned](#_Trm00068) or "[definitely assigned](#_Trm00068) after true expression", then the state of *v* after *expr* is "[definitely assigned](#_Trm00068) after true expression".
  + Otherwise, if the state of *v* after *expr\_first* is "[definitely assigned](#_Trm00068) after false expression", and the state of *v* after *expr\_second* is "[definitely assigned](#_Trm00068) after false expression", then the state of *v* after *expr* is "[definitely assigned](#_Trm00068) after false expression".
  + Otherwise, the state of *v* after *expr* is not [definitely assigned](#_Trm00068).

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](#_Trm00068) in one of the embedded [statements](#_Trm00037) of an if statement but not in the other. In the if statement in [method](#_Trm00056) F, the variable i is [definitely assigned](#_Trm00068) 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](#_Trm00068) 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](#_Trm00195) of *v* before *expr\_first* is the same as the [definite assignment state](#_Trm00195) of *v* before *expr*.
* The [definite assignment state](#_Trm00195) of *v* before *expr\_second* is [definitely assigned](#_Trm00068) if the state of *v* after *expr\_first* is either [definitely assigned](#_Trm00068) or "[definitely assigned](#_Trm00068) after false expression". Otherwise, it is not [definitely assigned](#_Trm00068).
* The [definite assignment state](#_Trm00195)ment of *v* after *expr* is determined by:
  + If *expr\_first* is a constant expression with the value true, then the [definite assignment state](#_Trm00195) of *v* after *expr* is the same as the [definite assignment state](#_Trm00195) of *v* after *expr\_first*.
  + Otherwise, if the state of *v* after *expr\_first* is [definitely assigned](#_Trm00068), then the state of *v* after *expr* is [definitely assigned](#_Trm00068).
  + Otherwise, if the state of *v* after *expr\_second* is [definitely assigned](#_Trm00068), and the state of *v* after *expr\_first* is "[definitely assigned](#_Trm00068) after true expression", then the state of *v* after *expr* is [definitely assigned](#_Trm00068).
  + Otherwise, if the state of *v* after *expr\_second* is [definitely assigned](#_Trm00068) or "[definitely assigned](#_Trm00068) after false expression", then the state of *v* after *expr* is "[definitely assigned](#_Trm00068) after false expression".
  + Otherwise, if the state of *v* after *expr\_first* is "[definitely assigned](#_Trm00068) after true expression", and the state of *v* after *expr\_second* is "[definitely assigned](#_Trm00068) after true expression", then the state of *v* after *expr* is "[definitely assigned](#_Trm00068) after true expression".
  + Otherwise, the state of *v* after *expr* is not [definitely assigned](#_Trm00068).

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](#_Trm00068) in one of the embedded [statements](#_Trm00037) of an if statement but not in the other. In the if statement in [method](#_Trm00056) G, the variable i is [definitely assigned](#_Trm00068) 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](#_Trm00068) 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](#_Trm00195) of *v* before *expr\_operand* is the same as the [definite assignment state](#_Trm00195) of *v* before *expr*.
* The [definite assignment state](#_Trm00195) of *v* after *expr* is determined by:
  + If the state of *v* after *expr\_operand* is [definitely assigned](#_Trm00068), then the state of *v* after *expr* is [definitely assigned](#_Trm00068).
  + If the state of *v* after *expr\_operand* is not [definitely assigned](#_Trm00068), then the state of *v* after *expr* is not [definitely assigned](#_Trm00068).
  + If the state of *v* after *expr\_operand* is "[definitely assigned](#_Trm00068) after false expression", then the state of *v* after *expr* is "[definitely assigned](#_Trm00068) after true expression".
  + If the state of *v* after *expr\_operand* is "[definitely assigned](#_Trm00068) after true expression", then the state of *v* after *expr* is "[definitely assigned](#_Trm00068) after false expression".

#### ?? (null coalescing) expressions

For an expression *expr* of the form expr\_first ?? expr\_second:

* The [definite assignment state](#_Trm00195) of *v* before *expr\_first* is the same as the [definite assignment state](#_Trm00195) of *v* before *expr*.
* The [definite assignment state](#_Trm00195) of *v* before *expr\_second* is the same as the [definite assignment state](#_Trm00195) of *v* after *expr\_first*.
* The [definite assignment state](#_Trm00195)ment of *v* after *expr* is determined by:
  + If *expr\_first* is a constant expression ([§7.19](#_Toc00346)) 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](#_Trm00195) of *v* after *expr\_first*.

#### ?: (conditional) expressions

For an expression *expr* of the form expr\_cond ? expr\_true : expr\_false:

* The [definite assignment state](#_Trm00195) of *v* before *expr\_cond* is the same as the state of *v* before *expr*.
* The [definite assignment state](#_Trm00195) of *v* before *expr\_true* is [definitely assigned](#_Trm00068) 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](#_Trm00068) or "[definitely assigned](#_Trm00068) after true expression".
* The [definite assignment state](#_Trm00195) of *v* before *expr\_false* is [definitely assigned](#_Trm00068) 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](#_Trm00068) or "[definitely assigned](#_Trm00068) after false expression".
* The [definite assignment state](#_Trm00195) of *v* after *expr* is determined by:
  + If *expr\_cond* is a constant expression ([§7.19](#_Toc00346)) 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 ([§7.19](#_Toc00346)) 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](#_Trm00068) and the state of *v* after *expr\_false* is [definitely assigned](#_Trm00068), then the state of *v* after *expr* is [definitely assigned](#_Trm00068).
  + Otherwise, the state of *v* after *expr* is not [definitely assigned](#_Trm00068).

#### Anonymous functions

For a [lambda\_expression](#_Grm00064) or [anonymous\_method\_expression](#_Grm00064) *expr* with a body (either [block](#_Grm00071) or [expression](#_Grm00067)) *body*:

* The [definite assignment state](#_Trm00195) of an outer variable *v* before *body* is the same as the state of *v* before *expr*. That is, [definite assignment state](#_Trm00195) of outer [variables](#_Trm00031) is [inherited](#_Trm00136) from the context of the anonymous function.
* The [definite assignment state](#_Trm00195) 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](#_Trm00068) 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](#_Trm00195) of n outside the anonymous function.

## Variable [references](#_Trm00160)

A [variable\_reference](#_Grm00033) is an [expression](#_Grm00067) that is classified as a variable. A [variable\_reference](#_Grm00033) 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](#_Grm00033) is known as an *lvalue*.

## Atomicity of variable [references](#_Trm00160)

Reads and writes of the following data [types](#_Trm00011) are atomic: bool, char, byte, sbyte, short, ushort, uint, int, float, and [reference types](#_Trm00019). In addition, reads and writes of [enum type](#_Trm00105)s with an [underlying type](#_Trm00106) in the previous list are also atomic. Reads and writes of other [types](#_Trm00011), including long, ulong, double, and decimal, as well as user-[defined](#_Trm00121) [types](#_Trm00011), 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

A ***conversion*** enables an expression to be treated as being of a particular type. A [conversion](#_Trm00196) 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](#_Trm00198) cast is required. For [instance](#_Trm00172), the [conversion](#_Trm00196) from type int to type long is [implicit](#_Trm00197), so expressions of type int can [implicit](#_Trm00197)ly be treated as type long. The opposite [conversion](#_Trm00196), from type long to type int, is [explicit](#_Trm00198) and so an [explicit](#_Trm00198) 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 [conversion](#_Trm00196)s are [defined](#_Trm00121) by the language. Programs may also define their own [conversion](#_Trm00196)s ([§6.4](#_Toc00193)).

## Implicit [conversion](#_Trm00196)s

The following [conversion](#_Trm00196)s are classified as [implicit](#_Trm00197) [conversion](#_Trm00196)s:

* Identity [conversion](#_Trm00196)s
* Implicit numeric [conversion](#_Trm00196)s
* Implicit enumeration [conversion](#_Trm00196)s.
* Implicit nullable [conversion](#_Trm00196)s
* Null [literal](#_Trm00118) [conversion](#_Trm00196)s
* Implicit reference [conversion](#_Trm00196)s
* Boxing [conversion](#_Trm00196)s
* Implicit dynamic [conversion](#_Trm00196)s
* Implicit constant expression [conversion](#_Trm00196)s
* User-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s
* Anonymous function [conversion](#_Trm00196)s
* Method group [conversion](#_Trm00196)s

Implicit [conversion](#_Trm00196)s can occur in a variety of situations, including function member invocations ([§7.5.4](#_Toc00249)), cast expressions ([§7.7.6](#_Toc00287)), and assignments ([§7.17](#_Toc00341)).

The pre-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s always succeed and never cause exceptions to be thrown. Properly designed user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s should exhibit these characteristics as well.

For the purposes of [conversion](#_Trm00196), the [types](#_Trm00011) object and dynamic are considered equivalent.

However, dynamic [conversion](#_Trm00196)s ([§6.1.8](#_Toc00176) and [§6.2.6](#_Toc00187)) apply only to expressions of type dynamic ([§4.2.3](#_Toc00105)).

### Identity [conversion](#_Trm00196)

An identity [conversion](#_Trm00196) converts from any type to the same type. This [conversion](#_Trm00196) exists such that an entity that already has a required type can be said to be convertible to that type.

* Because [object](#_Trm00173) and dynamic are considered equivalent there is an identity [conversion](#_Trm00196) between object and dynamic, and between [constructed type](#_Trm00178)s that are the same when replacing all occurences of dynamic with object.

### Implicit numeric [conversion](#_Trm00196)s

The [implicit](#_Trm00197) numeric [conversion](#_Trm00196)s 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](#_Trm00197) numeric [conversion](#_Trm00196)s never lose any information.

There are no [implicit](#_Trm00197) [conversion](#_Trm00196)s to the char type, so values of the other integral [types](#_Trm00011) do not automatically convert to the char type.

### Implicit enumeration [conversion](#_Trm00196)s

An [implicit](#_Trm00197) enumeration [conversion](#_Trm00196) permits the [decimal\_integer\_literal](#_Grm00011) 0 to be converted to any [enum\_type](#_Grm00029) and to any [nullable\_type](#_Grm00029) whose [underlying type](#_Trm00106) is an [enum\_type](#_Grm00029). In the latter case the [conversion](#_Trm00196) is evaluated by converting to the underlying [enum\_type](#_Grm00029) and [wrapping](#_Trm00171) the result ([§4.1.10](#_Toc00101)).

### Implicit nullable [conversion](#_Trm00196)s

Pre[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s that operate on [non-nullable value type](#_Trm00169)s can also be used with nullable forms of those [types](#_Trm00011). For each of the pre[defined](#_Trm00121) [implicit](#_Trm00197) identity and numeric [conversion](#_Trm00196)s that convert from a [non-nullable value type](#_Trm00169) S to a [non-nullable value type](#_Trm00169) T, the following [implicit](#_Trm00197) nullable [conversion](#_Trm00196)s exist:

* An [implicit](#_Trm00197) [conversion](#_Trm00196) from S? to T?.
* An [implicit](#_Trm00197) [conversion](#_Trm00196) from S to T?.

Evaluation of an [implicit](#_Trm00197) nullable [conversion](#_Trm00196) based on an underlying [conversion](#_Trm00196) from S to T proceeds as follows:

* If the nullable [conversion](#_Trm00196) is from S? to T?:
  + If the source value is null (HasValue property is false), the result is the [null value](#_Trm00165) of type T?.
  + Otherwise, the [conversion](#_Trm00196) is evaluated as an [unwrapping](#_Trm00170) from S? to S, followed by the underlying [conversion](#_Trm00196) from S to T, followed by a [wrapping](#_Trm00171) ([§4.1.10](#_Toc00101)) from T to T?.
* If the nullable [conversion](#_Trm00196) is from S to T?, the [conversion](#_Trm00196) is evaluated as the underlying [conversion](#_Trm00196) from S to T followed by a [wrapping](#_Trm00171) from T to T?.

### Null [literal](#_Trm00118) [conversion](#_Trm00196)s

An [implicit](#_Trm00197) [conversion](#_Trm00196) exists from the null [literal](#_Trm00118) to any nullable type. This [conversion](#_Trm00196) produces the [null value](#_Trm00165) ([§4.1.10](#_Toc00101)) of the given nullable type.

### Implicit reference [conversion](#_Trm00196)s

The [implicit](#_Trm00197) reference [conversion](#_Trm00196)s are:

* From any [reference\_type](#_Grm00030) to object and dynamic.
* From any [class\_type](#_Grm00030) S to any [class\_type](#_Grm00030) T, provided S is derived from T.
* From any [class\_type](#_Grm00030) S to any [interface\_type](#_Grm00030) T, provided S implements T.
* From any [interface\_type](#_Grm00030) S to any [interface\_type](#_Grm00030) T, provided S is derived from T.
* From an [array\_type](#_Grm00030) S with an [element type](#_Trm00095) SE to an [array\_type](#_Grm00030) T with an [element type](#_Trm00095) TE, provided all of the following are true:
  + S and T differ only in [element type](#_Trm00095). In other words, S and T have the same number of dimensions.
  + Both SE and TE are [reference\_type](#_Grm00030)s.
  + An [implicit](#_Trm00197) reference [conversion](#_Trm00196) exists from SE to TE.
* From any [array\_type](#_Grm00030) to System.Array and the [interface](#_Trm00102)s it implements.
* From a single-dimensional [array](#_Trm00093) type S[] to System.Collections.Generic.IList<T> and its base [interface](#_Trm00102)s, provided that there is an [implicit](#_Trm00197) identity or reference [conversion](#_Trm00196) from S to T.
* From any [delegate\_type](#_Grm00030) to System.Delegate and the [interface](#_Trm00102)s it implements.
* From the null [literal](#_Trm00118) to any [reference\_type](#_Grm00030).
* From any [reference\_type](#_Grm00030) to a [reference\_type](#_Grm00030) T if it has an [implicit](#_Trm00197) identity or reference [conversion](#_Trm00196) to a [reference\_type](#_Grm00030) T0 and T0 has an identity [conversion](#_Trm00196) to T.
* From any [reference\_type](#_Grm00030) to an [interface](#_Trm00102) or [delegate type](#_Trm00107) T if it has an [implicit](#_Trm00197) identity or reference [conversion](#_Trm00196) to an [interface](#_Trm00102) or [delegate type](#_Trm00107) T0 and T0 is variance-convertible ([§13.1.3.2](#_Toc00533)) to T.
* Implicit [conversion](#_Trm00196)s involving type [parameters](#_Trm00059) that are known to be [reference types](#_Trm00019). See [§6.1.10](#_Toc00178) for more details on [implicit](#_Trm00197) [conversion](#_Trm00196)s involving type [parameters](#_Trm00059).

The [implicit](#_Trm00197) reference [conversion](#_Trm00196)s are those [conversion](#_Trm00196)s between [reference\_type](#_Grm00030)s that can be proven to always succeed, and therefore require no checks at run-time.

Reference [conversion](#_Trm00196)s, [implicit](#_Trm00197) or [explicit](#_Trm00198), never change the referential identity of the [object](#_Trm00173) being converted. In other words, while a reference [conversion](#_Trm00196) may change the type of the reference, it never changes the type or value of the [object](#_Trm00173) being referred to.

### Boxing [conversion](#_Trm00196)s

A [boxing](#_Trm00029) [conversion](#_Trm00196) permits a [value\_type](#_Grm00029) to be [implicit](#_Trm00197)ly converted to a reference type. A [boxing](#_Trm00029) [conversion](#_Trm00196) exists from any [non\_nullable\_value\_type](#_Grm00029) to object and dynamic, to System.ValueType and to any [interface\_type](#_Grm00030) implemented by the [non\_nullable\_value\_type](#_Grm00029). Furthermore an [enum\_type](#_Grm00029) can be converted to the type System.Enum.

A [boxing](#_Trm00029) [conversion](#_Trm00196) exists from a [nullable\_type](#_Grm00029) to a reference type, if and only if a [boxing](#_Trm00029) [conversion](#_Trm00196) exists from the underlying [non\_nullable\_value\_type](#_Grm00029) to the reference type.

A value type has a [boxing](#_Trm00029) [conversion](#_Trm00196) to an [interface](#_Trm00102) type I if it has a [boxing](#_Trm00029) [conversion](#_Trm00196) to an [interface](#_Trm00102) type I0 and I0 has an identity [conversion](#_Trm00196) to I.

A value type has a [boxing](#_Trm00029) [conversion](#_Trm00196) to an [interface](#_Trm00102) type I if it has a [boxing](#_Trm00029) [conversion](#_Trm00196) to an [interface](#_Trm00102) or [delegate type](#_Trm00107) I0 and I0 is variance-convertible ([§13.1.3.2](#_Toc00533)) to I.

Boxing a value of a [non\_nullable\_value\_type](#_Grm00029) consists of allocating an [object](#_Trm00173) [instance](#_Trm00172) and copying the [value\_type](#_Grm00029) value into that [instance](#_Trm00172). A struct can be boxed to the type System.ValueType, since that is a base class for all [structs](#_Trm00092) ([§11.3.2](#_Toc00506)).

Boxing a value of a [nullable\_type](#_Grm00029) 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](#_Trm00170) and [boxing](#_Trm00029) the source value.

Boxing [conversion](#_Trm00196)s are described further in [§4.3.1](#_Toc00111).

### Implicit dynamic [conversion](#_Trm00196)s

An [implicit](#_Trm00197) dynamic [conversion](#_Trm00196) exists from an expression of type dynamic to any type T. The [conversion](#_Trm00196) is dynamically bound ([§7.2.2](#_Toc00209)), which means that an [implicit](#_Trm00197) [conversion](#_Trm00196) will be sought at run-time from the [run-time type](#_Trm00073) of the expression to T. If no [conversion](#_Trm00196) is found, a run-time exception is thrown.

Note that this [implicit](#_Trm00197) [conversion](#_Trm00196) seemingly violates the advice in the beginning of [§6.1](#_Toc00168) that an [implicit](#_Trm00197) [conversion](#_Trm00196) should never cause an exception. However it is not the [conversion](#_Trm00196) itself, but the *finding* of the [conversion](#_Trm00196) that causes the exception. The risk of run-time exceptions is inherent in the use of dynamic binding. If dynamic binding of the [conversion](#_Trm00196) is not desired, the expression can be first converted to object, and then to the desired type.

The following example illustrates [implicit](#_Trm00197) dynamic [conversion](#_Trm00196)s:

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](#_Trm00197) dynamic [conversion](#_Trm00196)s, where the binding of the operations is suspended until run-time. At run-time, [implicit](#_Trm00197) [conversion](#_Trm00196)s are sought from the [run-time type](#_Trm00073) of d -- string -- to the target type. A [conversion](#_Trm00196) is found to string but not to int.

### Implicit constant expression [conversion](#_Trm00196)s

An [implicit](#_Trm00197) constant expression [conversion](#_Trm00196) permits the following [conversion](#_Trm00196)s:

* A [constant\_expression](#_Grm00068) ([§7.19](#_Toc00346)) of type int can be converted to type sbyte, byte, short, ushort, uint, or ulong, provided the value of the [constant\_expression](#_Grm00068) is within the range of the destination type.
* A [constant\_expression](#_Grm00068) of type long can be converted to type ulong, provided the value of the [constant\_expression](#_Grm00068) is not negative.

### Implicit [conversion](#_Trm00196)s involving type [parameters](#_Trm00059)

The following [implicit](#_Trm00197) [conversion](#_Trm00196)s 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](#_Trm00102) implemented by C. At run-time, if T is a value type, the [conversion](#_Trm00196) is executed as a [boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [implicit](#_Trm00197) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).
* From T to an [interface](#_Trm00102) type I in T's effective [interface](#_Trm00102) set and from T to any base [interface](#_Trm00102) of I. At run-time, if T is a value type, the [conversion](#_Trm00196) is executed as a [boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [implicit](#_Trm00197) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).
* From T to a type parameter U, provided T depends on U ([§10.1.5](#_Toc00399)). At run-time, if U is a value type, then T and U are necessarily the same type and no [conversion](#_Trm00196) is performed. Otherwise, if T is a value type, the [conversion](#_Trm00196) is executed as a [boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [implicit](#_Trm00197) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).
* From the null [literal](#_Trm00118) to T, provided T is known to be a reference type.
* From T to a reference type I if it has an [implicit](#_Trm00197) [conversion](#_Trm00196) to a reference type S0 and S0 has an identity [conversion](#_Trm00196) to S. At run-time the [conversion](#_Trm00196) is executed the same way as the [conversion](#_Trm00196) to S0.
* From T to an [interface](#_Trm00102) type I if it has an [implicit](#_Trm00197) [conversion](#_Trm00196) to an [interface](#_Trm00102) or [delegate type](#_Trm00107) I0 and I0 is variance-convertible to I ([§13.1.3.2](#_Toc00533)). At run-time, if T is a value type, the [conversion](#_Trm00196) is executed as a [boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [implicit](#_Trm00197) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).

If T is known to be a reference type ([§10.1.5](#_Toc00399)), the [conversion](#_Trm00196)s above are all classified as [implicit](#_Trm00197) reference [conversion](#_Trm00196)s ([§6.1.6](#_Toc00174)). If T is not known to be a reference type, the [conversion](#_Trm00196)s above are classified as [boxing](#_Trm00029) [conversion](#_Trm00196)s ([§6.1.7](#_Toc00175)).

### User-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s

A user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196) consists of an optional standard [implicit](#_Trm00197) [conversion](#_Trm00196), followed by execution of a user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196) [operator](#_Trm00090), followed by another optional standard [implicit](#_Trm00197) [conversion](#_Trm00196). The exact rules for evaluating user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s are described in [§6.4.4](#_Toc00197).

### Anonymous function [conversion](#_Trm00196)s and [method](#_Trm00056) group [conversion](#_Trm00196)s

Anonymous functions and [method](#_Trm00056) groups do not have [types](#_Trm00011) in and of themselves, but may be [implicit](#_Trm00197)ly converted to [delegate type](#_Trm00107)s or [expression tree types](#_Trm00184). Anonymous function [conversion](#_Trm00196)s are described in more detail in [§6.5](#_Toc00199) and [method](#_Trm00056) group [conversion](#_Trm00196)s in [§6.6](#_Toc00203).

## Explicit [conversion](#_Trm00196)s

The following [conversion](#_Trm00196)s are classified as [explicit](#_Trm00198) [conversion](#_Trm00196)s:

* All [implicit](#_Trm00197) [conversion](#_Trm00196)s.
* Explicit numeric [conversion](#_Trm00196)s.
* Explicit enumeration [conversion](#_Trm00196)s.
* Explicit nullable [conversion](#_Trm00196)s.
* Explicit reference [conversion](#_Trm00196)s.
* Explicit [interface](#_Trm00102) [conversion](#_Trm00196)s.
* Un[boxing](#_Trm00029) [conversion](#_Trm00196)s.
* Explicit dynamic [conversion](#_Trm00196)s
* User-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s.

Explicit [conversion](#_Trm00196)s can occur in cast expressions ([§7.7.6](#_Toc00287)).

The set of [explicit](#_Trm00198) [conversion](#_Trm00196)s includes all [implicit](#_Trm00197) [conversion](#_Trm00196)s. This means that redundant cast expressions are allowed.

The [explicit](#_Trm00198) [conversion](#_Trm00196)s that are not [implicit](#_Trm00197) [conversion](#_Trm00196)s are [conversion](#_Trm00196)s that cannot be proven to always succeed, [conversion](#_Trm00196)s that are known to possibly lose information, and [conversion](#_Trm00196)s across domains of [types](#_Trm00011) sufficiently different to merit [explicit](#_Trm00198) notation.

### Explicit numeric [conversion](#_Trm00196)s

The [explicit](#_Trm00198) numeric [conversion](#_Trm00196)s are the [conversion](#_Trm00196)s from a [numeric\_type](#_Grm00029) to another [numeric\_type](#_Grm00029) for which an [implicit](#_Trm00197) numeric [conversion](#_Trm00196) ([§6.1.2](#_Toc00170)) 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](#_Trm00198) [conversion](#_Trm00196)s include all [implicit](#_Trm00197) and [explicit](#_Trm00198) numeric [conversion](#_Trm00196)s, it is always possible to convert from any [numeric\_type](#_Grm00029) to any other [numeric\_type](#_Grm00029) using a cast expression ([§7.7.6](#_Toc00287)).

The [explicit](#_Trm00198) numeric [conversion](#_Trm00196)s possibly lose information or possibly cause exceptions to be thrown. An [explicit](#_Trm00198) numeric [conversion](#_Trm00196) is processed as follows:

* For a [conversion](#_Trm00196) from an integral type to another integral type, the processing depends on the overflow checking context ([§7.6.12](#_Toc00278)) in which the [conversion](#_Trm00196) takes place:
  + In a checked context, the [conversion](#_Trm00196) 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](#_Trm00196) 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](#_Trm00196) 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](#_Trm00196). If the resulting integral value is outside the range of the destination type, a System.OverflowException is thrown.
* For a [conversion](#_Trm00196) from float or double to an integral type, the processing depends on the overflow checking context ([§7.6.12](#_Toc00278)) in which the [conversion](#_Trm00196) takes place:
  + In a checked context, the [conversion](#_Trm00196) 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](#_Trm00196).
    - Otherwise, a System.OverflowException is thrown.
  + In an unchecked context, the [conversion](#_Trm00196) always succeeds, and proceeds as follows.
    - If the value of the operand is NaN or infinite, the result of the [conversion](#_Trm00196) 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](#_Trm00196).
    - Otherwise, the result of the [conversion](#_Trm00196) is an unspecified value of the destination type.
* For a [conversion](#_Trm00196) 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](#_Trm00196) 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 ([§4.1.7](#_Toc00098)). 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](#_Trm00196) from decimal to float or double, the decimal value is rounded to the nearest double or float value. While this [conversion](#_Trm00196) may lose precision, it never causes an exception to be thrown.

### Explicit enumeration [conversion](#_Trm00196)s

The [explicit](#_Trm00198) enumeration [conversion](#_Trm00196)s are:

* From sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal to any [enum\_type](#_Grm00029).
* From any [enum\_type](#_Grm00029) to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal.
* From any [enum\_type](#_Grm00029) to any other [enum\_type](#_Grm00029).

An [explicit](#_Trm00198) enumeration [conversion](#_Trm00196) between two [types](#_Trm00011) is processed by treating any participating [enum\_type](#_Grm00029) as the [underlying type](#_Trm00106) of that [enum\_type](#_Grm00029), and then performing an [implicit](#_Trm00197) or [explicit](#_Trm00198) numeric [conversion](#_Trm00196) between the resulting [types](#_Trm00011). For example, given an [enum\_type](#_Grm00029) E with and [underlying type](#_Trm00106) of int, a [conversion](#_Trm00196) from E to byte is processed as an [explicit](#_Trm00198) numeric [conversion](#_Trm00196) ([§6.2.1](#_Toc00182)) from int to byte, and a [conversion](#_Trm00196) from byte to E is processed as an [implicit](#_Trm00197) numeric [conversion](#_Trm00196) ([§6.1.2](#_Toc00170)) from byte to int.

### Explicit nullable [conversion](#_Trm00196)s

***Explicit nullable conversions*** permit pre[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s that operate on [non-nullable value type](#_Trm00169)s to also be used with nullable forms of those [types](#_Trm00011). For each of the pre[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s that convert from a [non-nullable value type](#_Trm00169) S to a [non-nullable value type](#_Trm00169) T ([§6.1.1](#_Toc00169), [§6.1.2](#_Toc00170), [§6.1.3](#_Toc00171), [§6.2.1](#_Toc00182), and [§6.2.2](#_Toc00183)), the following nullable [conversion](#_Trm00196)s exist:

* An [explicit](#_Trm00198) [conversion](#_Trm00196) from S? to T?.
* An [explicit](#_Trm00198) [conversion](#_Trm00196) from S to T?.
* An [explicit](#_Trm00198) [conversion](#_Trm00196) from S? to T.

Evaluation of a nullable [conversion](#_Trm00196) based on an underlying [conversion](#_Trm00196) from S to T proceeds as follows:

* If the nullable [conversion](#_Trm00196) is from S? to T?:
  + If the source value is null (HasValue property is false), the result is the [null value](#_Trm00165) of type T?.
  + Otherwise, the [conversion](#_Trm00196) is evaluated as an [unwrapping](#_Trm00170) from S? to S, followed by the underlying [conversion](#_Trm00196) from S to T, followed by a [wrapping](#_Trm00171) from T to T?.
* If the nullable [conversion](#_Trm00196) is from S to T?, the [conversion](#_Trm00196) is evaluated as the underlying [conversion](#_Trm00196) from S to T followed by a [wrapping](#_Trm00171) from T to T?.
* If the nullable [conversion](#_Trm00196) is from S? to T, the [conversion](#_Trm00196) is evaluated as an [unwrapping](#_Trm00170) from S? to S followed by the underlying [conversion](#_Trm00196) from S to T.

Note that an attempt to unwrap a nullable value will throw an exception if the value is null.

### Explicit reference [conversion](#_Trm00196)s

The [explicit](#_Trm00198) reference [conversion](#_Trm00196)s are:

* From object and dynamic to any other [reference\_type](#_Grm00030).
* From any [class\_type](#_Grm00030) S to any [class\_type](#_Grm00030) T, provided S is a base class of T.
* From any [class\_type](#_Grm00030) S to any [interface\_type](#_Grm00030) T, provided S is not sealed and provided S does not implement T.
* From any [interface\_type](#_Grm00030) S to any [class\_type](#_Grm00030) T, provided T is not sealed or provided T implements S.
* From any [interface\_type](#_Grm00030) S to any [interface\_type](#_Grm00030) T, provided S is not derived from T.
* From an [array\_type](#_Grm00030) S with an [element type](#_Trm00095) SE to an [array\_type](#_Grm00030) T with an [element type](#_Trm00095) TE, provided all of the following are true:
  + S and T differ only in [element type](#_Trm00095). In other words, S and T have the same number of dimensions.
  + Both SE and TE are [reference\_type](#_Grm00030)s.
  + An [explicit](#_Trm00198) reference [conversion](#_Trm00196) exists from SE to TE.
* From System.Array and the [interface](#_Trm00102)s it implements to any [array\_type](#_Grm00030).
* From a single-dimensional [array](#_Trm00093) type S[] to System.Collections.Generic.IList<T> and its base [interface](#_Trm00102)s, provided that there is an [explicit](#_Trm00198) reference [conversion](#_Trm00196) from S to T.
* From System.Collections.Generic.IList<S> and its base [interface](#_Trm00102)s to a single-dimensional [array](#_Trm00093) type T[], provided that there is an [explicit](#_Trm00198) identity or reference [conversion](#_Trm00196) from S to T.
* From System.Delegate and the [interface](#_Trm00102)s it implements to any [delegate\_type](#_Grm00030).
* From a reference type to a reference type T if it has an [explicit](#_Trm00198) reference [conversion](#_Trm00196) to a reference type T0 and T0 has an identity [conversion](#_Trm00196) T.
* From a reference type to an [interface](#_Trm00102) or [delegate type](#_Trm00107) T if it has an [explicit](#_Trm00198) reference [conversion](#_Trm00196) to an [interface](#_Trm00102) or [delegate type](#_Trm00107) T0 and either T0 is variance-convertible to T or T is variance-convertible to T0 ([§13.1.3.2](#_Toc00533)).
* From D<S1...Sn> to D<T1...Tn> where D<X1...Xn> is a generic [delegate type](#_Trm00107), 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](#_Trm00197) or [explicit](#_Trm00198) identity or reference [conversion](#_Trm00196) from Si to Ti.
  + If Xi is contravariant, then Si and Ti are either identical or both [reference types](#_Trm00019).
* Explicit [conversion](#_Trm00196)s involving type [parameters](#_Trm00059) that are known to be [reference types](#_Trm00019). For more details on [explicit](#_Trm00198) [conversion](#_Trm00196)s involving type [parameters](#_Trm00059), see [§6.2.7](#_Toc00188).

The [explicit](#_Trm00198) reference [conversion](#_Trm00196)s are those [conversion](#_Trm00196)s between reference-[types](#_Trm00011) that require run-time checks to ensure they are correct.

For an [explicit](#_Trm00198) reference [conversion](#_Trm00196) to succeed at run-time, the value of the source operand must be null, or the actual type of the [object](#_Trm00173) referenced by the source operand must be a type that can be converted to the destination type by an [implicit](#_Trm00197) reference [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) or [boxing](#_Trm00029) [conversion](#_Trm00196) ([§6.1.7](#_Toc00175)). If an [explicit](#_Trm00198) reference [conversion](#_Trm00196) fails, a System.InvalidCastException is thrown.

Reference [conversion](#_Trm00196)s, [implicit](#_Trm00197) or [explicit](#_Trm00198), never change the referential identity of the [object](#_Trm00173) being converted. In other words, while a reference [conversion](#_Trm00196) may change the type of the reference, it never changes the type or value of the [object](#_Trm00173) being referred to.

### Un[boxing](#_Trm00029) [conversion](#_Trm00196)s

An un[boxing](#_Trm00029) [conversion](#_Trm00196) permits a reference type to be [explicit](#_Trm00198)ly converted to a [value\_type](#_Grm00029). An un[boxing](#_Trm00029) [conversion](#_Trm00196) exists from the [types](#_Trm00011) object, dynamic and System.ValueType to any [non\_nullable\_value\_type](#_Grm00029), and from any [interface\_type](#_Grm00030) to any [non\_nullable\_value\_type](#_Grm00029) that implements the [interface\_type](#_Grm00030). Furthermore type System.Enum can be unboxed to any [enum\_type](#_Grm00029).

An un[boxing](#_Trm00029) [conversion](#_Trm00196) exists from a reference type to a [nullable\_type](#_Grm00029) if an un[boxing](#_Trm00029) [conversion](#_Trm00196) exists from the reference type to the underlying [non\_nullable\_value\_type](#_Grm00029) of the [nullable\_type](#_Grm00029).

A value type S has an un[boxing](#_Trm00029) [conversion](#_Trm00196) from an [interface](#_Trm00102) type I if it has an un[boxing](#_Trm00029) [conversion](#_Trm00196) from an [interface](#_Trm00102) type I0 and I0 has an identity [conversion](#_Trm00196) to I.

A value type S has an un[boxing](#_Trm00029) [conversion](#_Trm00196) from an [interface](#_Trm00102) type I if it has an un[boxing](#_Trm00029) [conversion](#_Trm00196) from an [interface](#_Trm00102) or [delegate type](#_Trm00107) I0 and either I0 is variance-convertible to I or I is variance-convertible to I0 ([§13.1.3.2](#_Toc00533)).

An un[boxing](#_Trm00029) operation consists of first checking that the [object](#_Trm00173) [instance](#_Trm00172) is a boxed value of the given [value\_type](#_Grm00029), and then copying the value out of the [instance](#_Trm00172). Un[boxing](#_Trm00029) a null reference to a [nullable\_type](#_Grm00029) produces the [null value](#_Trm00165) of the [nullable\_type](#_Grm00029). A struct can be unboxed from the type System.ValueType, since that is a base class for all [structs](#_Trm00092) ([§11.3.2](#_Toc00506)).

Un[boxing](#_Trm00029) [conversion](#_Trm00196)s are described further in [§4.3.2](#_Toc00112).

### Explicit dynamic [conversion](#_Trm00196)s

An [explicit](#_Trm00198) dynamic [conversion](#_Trm00196) exists from an expression of type dynamic to any type T. The [conversion](#_Trm00196) is dynamically bound ([§7.2.2](#_Toc00209)), which means that an [explicit](#_Trm00198) [conversion](#_Trm00196) will be sought at run-time from the [run-time type](#_Trm00073) of the expression to T. If no [conversion](#_Trm00196) is found, a run-time exception is thrown.

If dynamic binding of the [conversion](#_Trm00196) is not desired, the expression can be first converted to object, and then to the desired type.

Assume the following class is [defined](#_Trm00121):

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](#_Trm00198) dynamic [conversion](#_Trm00196)s:

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](#_Trm00196) of o to C is found at compile-time to be an [explicit](#_Trm00198) reference [conversion](#_Trm00196). This fails at run-time, because "1" is not in fact a C. The [conversion](#_Trm00196) of d to C however, as an [explicit](#_Trm00198) dynamic [conversion](#_Trm00196), is suspended to run-time, where a user [defined](#_Trm00121) [conversion](#_Trm00196) from the [run-time type](#_Trm00073) of d -- string -- to C is found, and succeeds.

### Explicit [conversion](#_Trm00196)s involving type [parameters](#_Trm00059)

The following [explicit](#_Trm00198) [conversion](#_Trm00196)s 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](#_Trm00196) is executed as an un[boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [explicit](#_Trm00198) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).
* From any [interface](#_Trm00102) type to T. At run-time, if T is a value type, the [conversion](#_Trm00196) is executed as an un[boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [explicit](#_Trm00198) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).
* From T to any [interface\_type](#_Grm00030) I provided there is not already an [implicit](#_Trm00197) [conversion](#_Trm00196) from T to I. At run-time, if T is a value type, the [conversion](#_Trm00196) is executed as a [boxing](#_Trm00029) [conversion](#_Trm00196) followed by an [explicit](#_Trm00198) reference [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [explicit](#_Trm00198) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).
* From a type parameter U to T, provided T depends on U ([§10.1.5](#_Toc00399)). At run-time, if U is a value type, then T and U are necessarily the same type and no [conversion](#_Trm00196) is performed. Otherwise, if T is a value type, the [conversion](#_Trm00196) is executed as an un[boxing](#_Trm00029) [conversion](#_Trm00196). Otherwise, the [conversion](#_Trm00196) is executed as an [explicit](#_Trm00198) reference [conversion](#_Trm00196) or identity [conversion](#_Trm00196).

If T is known to be a reference type, the [conversion](#_Trm00196)s above are all classified as [explicit](#_Trm00198) reference [conversion](#_Trm00196)s ([§6.2.4](#_Toc00185)). If T is not known to be a reference type, the [conversion](#_Trm00196)s above are classified as un[boxing](#_Trm00029) [conversion](#_Trm00196)s ([§6.2.5](#_Toc00186)).

The above rules do not permit a direct [explicit](#_Trm00198) [conversion](#_Trm00196) from an unconstrained type parameter to a non-[interface](#_Trm00102) type, which might be surprising. The reason for this rule is to pr[event](#_Trm00088) confusion and make the semantics of such [conversion](#_Trm00196)s clear. For example, consider the following declaration:

class X<T>  
{  
 public static long F(T t) {  
 return (long)t; // Error  
 }  
}

If the direct [explicit](#_Trm00198) [conversion](#_Trm00196) 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 [conversion](#_Trm00196)s are only considered when the [types](#_Trm00011) 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](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s

A user-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196) consists of an optional standard [explicit](#_Trm00198) [conversion](#_Trm00196), followed by execution of a user-[defined](#_Trm00121) [implicit](#_Trm00197) or [explicit](#_Trm00198) [conversion](#_Trm00196) [operator](#_Trm00090), followed by another optional standard [explicit](#_Trm00198) [conversion](#_Trm00196). The exact rules for evaluating user-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s are described in [§6.4.5](#_Toc00198).

## Standard [conversion](#_Trm00196)s

The standard [conversion](#_Trm00196)s are those pre-[defined](#_Trm00121) [conversion](#_Trm00196)s that can occur as part of a user-[defined](#_Trm00121) [conversion](#_Trm00196).

### Standard [implicit](#_Trm00197) [conversion](#_Trm00196)s

The following [implicit](#_Trm00197) [conversion](#_Trm00196)s are classified as standard [implicit](#_Trm00197) [conversion](#_Trm00196)s:

* Identity [conversion](#_Trm00196)s ([§6.1.1](#_Toc00169))
* Implicit numeric [conversion](#_Trm00196)s ([§6.1.2](#_Toc00170))
* Implicit nullable [conversion](#_Trm00196)s ([§6.1.4](#_Toc00172))
* Implicit reference [conversion](#_Trm00196)s ([§6.1.6](#_Toc00174))
* Boxing [conversion](#_Trm00196)s ([§6.1.7](#_Toc00175))
* Implicit constant expression [conversion](#_Trm00196)s ([§6.1.8](#_Toc00176))
* Implicit [conversion](#_Trm00196)s involving type [parameters](#_Trm00059) ([§6.1.10](#_Toc00178))

The standard [implicit](#_Trm00197) [conversion](#_Trm00196)s specifically exclude user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s.

### Standard [explicit](#_Trm00198) [conversion](#_Trm00196)s

The standard [explicit](#_Trm00198) [conversion](#_Trm00196)s are all standard [implicit](#_Trm00197) [conversion](#_Trm00196)s plus the subset of the [explicit](#_Trm00198) [conversion](#_Trm00196)s for which an opposite standard [implicit](#_Trm00197) [conversion](#_Trm00196) exists. In other words, if a standard [implicit](#_Trm00197) [conversion](#_Trm00196) exists from a type A to a type B, then a standard [explicit](#_Trm00198) [conversion](#_Trm00196) exists from type A to type B and from type B to type A.

## User-[defined](#_Trm00121) [conversion](#_Trm00196)s

C# allows the pre-[defined](#_Trm00121) [implicit](#_Trm00197) and [explicit](#_Trm00198) [conversion](#_Trm00196)s to be augmented by ***user-defined conversions***. User-[defined](#_Trm00121) [conversion](#_Trm00196)s are introduced by declaring [conversion](#_Trm00196) [operator](#_Trm00090)s ([§10.10.3](#_Toc00473)) in class and [struct types](#_Trm00022).

### Permitted user-[defined](#_Trm00121) [conversion](#_Trm00196)s

C# permits only certain user-[defined](#_Trm00121) [conversion](#_Trm00196)s to be declared. In particular, it is not possible to redefine an already existing [implicit](#_Trm00197) or [explicit](#_Trm00198) [conversion](#_Trm00196).

For a given source type S and target type T, if S or T are [nullable types](#_Trm00023), let S0 and T0 refer to their underlying [types](#_Trm00011), otherwise S0 and T0 are equal to S and T respectively. A class or struct is permitted to declare a [conversion](#_Trm00196) from a source type S to a target type T only if all of the following are true:

* S0 and T0 are different [types](#_Trm00011).
* Either S0 or T0 is the class or struct type in which the [operator](#_Trm00090) declaration takes place.
* Neither S0 nor T0 is an [interface\_type](#_Grm00030).
* Excluding user-[defined](#_Trm00121) [conversion](#_Trm00196)s, a [conversion](#_Trm00196) does not exist from S to T or from T to S.

The restrictions that apply to user-[defined](#_Trm00121) [conversion](#_Trm00196)s are discussed further in [§10.10.3](#_Toc00473).

### Lifted [conversion](#_Trm00196) [operator](#_Trm00090)s

Given a user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) that converts from a [non-nullable value type](#_Trm00169) S to a [non-nullable value type](#_Trm00169) T, a ***lifted conversion operator*** exists that converts from S? to T?. This lifted [conversion](#_Trm00196) [operator](#_Trm00090) performs an [unwrapping](#_Trm00170) from S? to S followed by the user-[defined](#_Trm00121) [conversion](#_Trm00196) from S to T followed by a [wrapping](#_Trm00171) from T to T?, except that a [null value](#_Trm00165)d S? converts directly to a [null value](#_Trm00165)d T?.

A lifted [conversion](#_Trm00196) [operator](#_Trm00090) has the same [implicit](#_Trm00197) or [explicit](#_Trm00198) classification as its underlying user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090). The term "user-[defined](#_Trm00121) [conversion](#_Trm00196)" applies to the use of both user-[defined](#_Trm00121) and lifted [conversion](#_Trm00196) [operator](#_Trm00090)s.

### Evaluation of user-[defined](#_Trm00121) [conversion](#_Trm00196)s

A user-[defined](#_Trm00121) [conversion](#_Trm00196) converts a value from its type, called the ***source type***, to another type, called the ***target type***. Evaluation of a user-[defined](#_Trm00121) [conversion](#_Trm00196) centers on finding the ***most specific*** user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) for the particular source and [target type](#_Trm00203)s. This determination is broken into several steps:

* Finding the set of classes and [structs](#_Trm00092) from which user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090)s will be considered. This set consists of the [source type](#_Trm00202) and its [base classes](#_Trm00050) and the [target type](#_Trm00203) and its [base classes](#_Trm00050) (with the [implicit](#_Trm00197) assumptions that only classes and [structs](#_Trm00092) can declare user-[defined](#_Trm00121) [operator](#_Trm00090)s, and that non-[class types](#_Trm00024) have no [base classes](#_Trm00050)). For the purposes of this step, if either the source or [target type](#_Trm00203) is a [nullable\_type](#_Grm00029), their [underlying type](#_Trm00106) is used instead.
* From that set of [types](#_Trm00011), determining which user-[defined](#_Trm00121) and lifted [conversion](#_Trm00196) [operator](#_Trm00090)s are applicable. For a [conversion](#_Trm00196) [operator](#_Trm00090) to be applicable, it must be possible to perform a standard [conversion](#_Trm00196) ([§6.3](#_Toc00190)) from the [source type](#_Trm00202) to the operand type of the [operator](#_Trm00090), and it must be possible to perform a standard [conversion](#_Trm00196) from the result type of the [operator](#_Trm00090) to the [target type](#_Trm00203).
* From the set of applicable user-[defined](#_Trm00121) [operator](#_Trm00090)s, determining which [operator](#_Trm00090) is unambiguously the [most specific](#_Trm00204). In general terms, the [most specific](#_Trm00204) [operator](#_Trm00090) is the [operator](#_Trm00090) whose operand type is "closest" to the [source type](#_Trm00202) and whose result type is "closest" to the [target type](#_Trm00203). User-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090)s are preferred over lifted [conversion](#_Trm00196) [operator](#_Trm00090)s. The exact rules for establishing the [most specific](#_Trm00204) user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) are [defined](#_Trm00121) in the following sections.

Once a [most specific](#_Trm00204) user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) has been identified, the actual execution of the user-[defined](#_Trm00121) [conversion](#_Trm00196) involves up to three steps:

* First, if required, performing a standard [conversion](#_Trm00196) from the [source type](#_Trm00202) to the operand type of the user-[defined](#_Trm00121) or lifted [conversion](#_Trm00196) [operator](#_Trm00090).
* Next, invoking the user-[defined](#_Trm00121) or lifted [conversion](#_Trm00196) [operator](#_Trm00090) to perform the [conversion](#_Trm00196).
* Finally, if required, performing a standard [conversion](#_Trm00196) from the result type of the user-[defined](#_Trm00121) or lifted [conversion](#_Trm00196) [operator](#_Trm00090) to the [target type](#_Trm00203).

Evaluation of a user-[defined](#_Trm00121) [conversion](#_Trm00196) never involves more than one user-[defined](#_Trm00121) or lifted [conversion](#_Trm00196) [operator](#_Trm00090). In other words, a [conversion](#_Trm00196) from type S to type T will never first execute a user-[defined](#_Trm00121) [conversion](#_Trm00196) from S to X and then execute a user-[defined](#_Trm00121) [conversion](#_Trm00196) from X to T.

Exact definitions of evaluation of user-[defined](#_Trm00121) [implicit](#_Trm00197) or [explicit](#_Trm00198) [conversion](#_Trm00196)s are given in the following sections. The definitions make use of the following terms:

* If a standard [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.3.1](#_Toc00191)) exists from a type A to a type B, and if neither A nor B are [interface\_type](#_Grm00030)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](#_Trm00011) is the one type that [encompass](#_Trm00206)es all other [types](#_Trm00011) in the set. If no single type [encompass](#_Trm00206)es all other [types](#_Trm00011), then the set has no most [encompass](#_Trm00206)ing type. In more intuitive terms, the most [encompass](#_Trm00206)ing type is the "largest" type in the set—the one type to which each of the other [types](#_Trm00011) can be [implicit](#_Trm00197)ly converted.
* The ***most encompassed type*** in a set of [types](#_Trm00011) is the one type that is [encompass](#_Trm00206)ed by all other [types](#_Trm00011) in the set. If no single type is [encompass](#_Trm00206)ed by all other [types](#_Trm00011), then the set has no most [encompass](#_Trm00206)ed type. In more intuitive terms, the most [encompass](#_Trm00206)ed type is the "smallest" type in the set—the one type that can be [implicit](#_Trm00197)ly converted to each of the other [types](#_Trm00011).

### Processing of user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s

A user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196) from type S to type T is processed as follows:

* Determine the [types](#_Trm00011) S0 and T0. If S or T are [nullable types](#_Trm00023), S0 and T0 are their underlying [types](#_Trm00011), otherwise S0 and T0 are equal to S and T respectively.
* Find the set of [types](#_Trm00011), D, from which user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090)s will be considered. This set consists of S0 (if S0 is a class or struct), the [base classes](#_Trm00050) of S0 (if S0 is a class), and T0 (if T0 is a class or struct).
* Find the set of applicable user-[defined](#_Trm00121) and lifted [conversion](#_Trm00196) [operator](#_Trm00090)s, U. This set consists of the user-[defined](#_Trm00121) and lifted [implicit](#_Trm00197) [conversion](#_Trm00196) [operator](#_Trm00090)s declared by the classes or [structs](#_Trm00092) in D that convert from a type [encompass](#_Trm00206)ing S to a type [encompass](#_Trm00206)ed by T. If U is empty, the [conversion](#_Trm00196) is un[defined](#_Trm00121) and a compile-time error occurs.
* Find the [most specific](#_Trm00204) [source type](#_Trm00202), SX, of the [operator](#_Trm00090)s in U:
  + If any of the [operator](#_Trm00090)s in U convert from S, then SX is S.
  + Otherwise, SX is the most [encompass](#_Trm00206)ed type in the combined set of [source type](#_Trm00202)s of the [operator](#_Trm00090)s in U. If exactly one most [encompass](#_Trm00206)ed type cannot be found, then the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
* Find the [most specific](#_Trm00204) [target type](#_Trm00203), TX, of the [operator](#_Trm00090)s in U:
  + If any of the [operator](#_Trm00090)s in U convert to T, then TX is T.
  + Otherwise, TX is the most [encompass](#_Trm00206)ing type in the combined set of [target type](#_Trm00203)s of the [operator](#_Trm00090)s in U. If exactly one most [encompass](#_Trm00206)ing type cannot be found, then the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
* Find the [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090):
  + If U contains exactly one user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) that converts from SX to TX, then this is the [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090).
  + Otherwise, if U contains exactly one lifted [conversion](#_Trm00196) [operator](#_Trm00090) that converts from SX to TX, then this is the [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090).
  + Otherwise, the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
* Finally, apply the [conversion](#_Trm00196):
  + If S is not SX, then a standard [implicit](#_Trm00197) [conversion](#_Trm00196) from S to SX is performed.
  + The [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090) is invoked to convert from SX to TX.
  + If TX is not T, then a standard [implicit](#_Trm00197) [conversion](#_Trm00196) from TX to T is performed.

### Processing of user-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s

A user-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196) from type S to type T is processed as follows:

* Determine the [types](#_Trm00011) S0 and T0. If S or T are [nullable types](#_Trm00023), S0 and T0 are their underlying [types](#_Trm00011), otherwise S0 and T0 are equal to S and T respectively.
* Find the set of [types](#_Trm00011), D, from which user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090)s will be considered. This set consists of S0 (if S0 is a class or struct), the [base classes](#_Trm00050) of S0 (if S0 is a class), T0 (if T0 is a class or struct), and the [base classes](#_Trm00050) of T0 (if T0 is a class).
* Find the set of applicable user-[defined](#_Trm00121) and lifted [conversion](#_Trm00196) [operator](#_Trm00090)s, U. This set consists of the user-[defined](#_Trm00121) and lifted [implicit](#_Trm00197) or [explicit](#_Trm00198) [conversion](#_Trm00196) [operator](#_Trm00090)s declared by the classes or [structs](#_Trm00092) in D that convert from a type [encompass](#_Trm00206)ing or [encompass](#_Trm00206)ed by S to a type [encompass](#_Trm00206)ing or [encompass](#_Trm00206)ed by T. If U is empty, the [conversion](#_Trm00196) is un[defined](#_Trm00121) and a compile-time error occurs.
* Find the [most specific](#_Trm00204) [source type](#_Trm00202), SX, of the [operator](#_Trm00090)s in U:
  + If any of the [operator](#_Trm00090)s in U convert from S, then SX is S.
  + Otherwise, if any of the [operator](#_Trm00090)s in U convert from [types](#_Trm00011) that [encompass](#_Trm00206) S, then SX is the most [encompass](#_Trm00206)ed type in the combined set of [source type](#_Trm00202)s of those [operator](#_Trm00090)s. If no most [encompass](#_Trm00206)ed type can be found, then the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
  + Otherwise, SX is the most [encompass](#_Trm00206)ing type in the combined set of [source type](#_Trm00202)s of the [operator](#_Trm00090)s in U. If exactly one most [encompass](#_Trm00206)ing type cannot be found, then the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
* Find the [most specific](#_Trm00204) [target type](#_Trm00203), TX, of the [operator](#_Trm00090)s in U:
  + If any of the [operator](#_Trm00090)s in U convert to T, then TX is T.
  + Otherwise, if any of the [operator](#_Trm00090)s in U convert to [types](#_Trm00011) that are [encompass](#_Trm00206)ed by T, then TX is the most [encompass](#_Trm00206)ing type in the combined set of [target type](#_Trm00203)s of those [operator](#_Trm00090)s. If exactly one most [encompass](#_Trm00206)ing type cannot be found, then the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
  + Otherwise, TX is the most [encompass](#_Trm00206)ed type in the combined set of [target type](#_Trm00203)s of the [operator](#_Trm00090)s in U. If no most [encompass](#_Trm00206)ed type can be found, then the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
* Find the [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090):
  + If U contains exactly one user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) that converts from SX to TX, then this is the [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090).
  + Otherwise, if U contains exactly one lifted [conversion](#_Trm00196) [operator](#_Trm00090) that converts from SX to TX, then this is the [most specific](#_Trm00204) [conversion](#_Trm00196) [operator](#_Trm00090).
  + Otherwise, the [conversion](#_Trm00196) is ambiguous and a compile-time error occurs.
* Finally, apply the [conversion](#_Trm00196):
  + If S is not SX, then a standard [explicit](#_Trm00198) [conversion](#_Trm00196) from S to SX is performed.
  + The [most specific](#_Trm00204) user-[defined](#_Trm00121) [conversion](#_Trm00196) [operator](#_Trm00090) is invoked to convert from SX to TX.
  + If TX is not T, then a standard [explicit](#_Trm00198) [conversion](#_Trm00196) from TX to T is performed.

## Anonymous function [conversion](#_Trm00196)s

An [anonymous\_method\_expression](#_Grm00064) or [lambda\_expression](#_Grm00064) is classified as an anonymous function ([§7.15](#_Toc00321)). The expression does not have a type but can be [implicit](#_Trm00197)ly converted to a compatible [delegate type](#_Trm00107) or expression tree type. Specifically, an anonymous function F is compatible with a [delegate type](#_Trm00107) D provided:

* If F contains an [anonymous\_function\_signature](#_Grm00064), then D and F have the same number of [parameters](#_Trm00059).
* If F does not contain an [anonymous\_function\_signature](#_Grm00064), then D may have zero or more [parameters](#_Trm00059) of any type, as long as no parameter of D has the out parameter modifier.
* If F has an [explicit](#_Trm00198)ly typed parameter list, each parameter in D has the same type and modifiers as the corresponding parameter in F.
* If F has an [implicit](#_Trm00197)ly typed parameter list, D has no ref or out [parameters](#_Trm00059).
* If the body of F is an expression, and either D has a void [return type](#_Trm00060) or F is async and D has the [return type](#_Trm00060) 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 [§7](#_Toc00204)) that would be permitted as a [statement\_expression](#_Grm00078) ([§8.6](#_Toc00357)).
* If the body of F is a statement [block](#_Trm00038), and either D has a void [return type](#_Trm00060) or F is async and D has the [return type](#_Trm00060) 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](#_Trm00038) (wrt [§8.2](#_Toc00350)) 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](#_Trm00060) T, *or* F is async and D has a [return type](#_Trm00060) 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 [§7](#_Toc00204)) that is [implicit](#_Trm00197)ly convertible to T.
* If the body of F is a statement [block](#_Trm00038), and *either* F is non-async and D has a non-void [return type](#_Trm00060) T, *or* F is async and D has a [return type](#_Trm00060) 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](#_Trm00038) (wrt [§8.2](#_Toc00350)) with a non-reachable end point in which each return statement specifies an expression that is [implicit](#_Trm00197)ly convertible to T.

For the purpose of brevity, this section uses the short form for the task [types](#_Trm00011) Task and Task<T> ([§10.15](#_Toc00494)).

A lambda expression F is compatible with an expression tree type Expression<D> if F is compatible with the [delegate type](#_Trm00107) D. Note that this does not apply to anonymous [method](#_Trm00056)s, only lambda expressions.

Certain lambda expressions cannot be converted to [expression tree types](#_Trm00184): Even though the [conversion](#_Trm00196) *exists*, it fails at compile-time. This is the case if the lambda expression:

* Has a [block](#_Grm00071) body
* Contains simple or compound assignment [operator](#_Trm00090)s
* Contains a dynamically bound expression
* Is async

The examples that follow use a generic [delegate type](#_Trm00107) 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 type](#_Trm00060)s 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](#_Trm00107) Func<int,int> because, when x is given type int, x+1 is a valid expression that is [implicit](#_Trm00197)ly convertible to type int.

Likewise, the second assignment successfully converts the anonymous function to the [delegate type](#_Trm00107) Func<int,double> because the result of x+1 (of type int) is [implicit](#_Trm00197)ly 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 [implicit](#_Trm00197)ly convertible to type int.

The fourth assignment successfully converts the anonymous async function to the [delegate type](#_Trm00107) Func<int, Task<int>> because the result of x+1 (of type int) is [implicit](#_Trm00197)ly convertible to the result type int of the task type Task<int>.

Anonymous functions may influence [overload resolution](#_Trm00078), and participate in type inference. See [§7.5](#_Toc00223) for further details.

### Evaluation of anonymous function [conversion](#_Trm00196)s to [delegate type](#_Trm00107)s

Conversion of an anonymous function to a [delegate type](#_Trm00107) produces a delegate [instance](#_Trm00172) which [references](#_Trm00160) the anonymous function and the (possibly empty) set of captured outer [variables](#_Trm00031) 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](#_Trm00031) referenced by the delegate.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target [object](#_Trm00173) and target [method](#_Trm00056) of the delegate are unspecified. In particular, it is unspecified whether the target [object](#_Trm00173) of the delegate is null, the this value of the enclosing function member, or some other [object](#_Trm00173).

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable [instance](#_Trm00172)s to the same [delegate type](#_Trm00107)s are permitted (but not required) to return the same delegate [instance](#_Trm00172). 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](#_Trm00062). 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](#_Trm00031), and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target [method](#_Trm00056). Indeed, the compiler is permitted to return the very same delegate [instance](#_Trm00172) from both anonymous function expressions.

### Evaluation of anonymous function [conversion](#_Trm00196)s to [expression tree types](#_Trm00184)

Conversion of an anonymous function to an expression tree type produces an expression tree ([§4.6](#_Toc00119)). More precisely, evaluation of the anonymous function [conversion](#_Trm00196) leads to the construction of an [object](#_Trm00173) 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](#_Trm00121).

### Implementation example

This section describes a possible implementation of anonymous function [conversion](#_Trm00196)s in terms of other C# con[structs](#_Trm00092). 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 [conversion](#_Trm00196)s to expression trees, as their exact semantics are outside the [scope](#_Trm00148) 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# con[structs](#_Trm00092) is provided. In the examples, the identifier D is assumed by represent the following [delegate type](#_Trm00107):

public delegate void D();

The simplest form of an anonymous function is one that captures no outer [variables](#_Trm00031):

class Test  
{  
 static void F() {  
 D d = () => { Console.WriteLine("test"); };  
 }  
}

This can be translated to a delegate instantiation that [references](#_Trm00160) a compiler generated static [method](#_Trm00056) 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](#_Trm00160) [instance](#_Trm00172) [members](#_Trm00012) of this:

class Test  
{  
 int x;  
  
 void F() {  
 D d = () => { Console.WriteLine(x); };  
 }  
}

This can be translated to a compiler generated [instance](#_Trm00172) [method](#_Trm00056) 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](#_Trm00193):

class Test  
{  
 void F() {  
 int y = 123;  
 D d = () => { Console.WriteLine(y); };  
 }  
}

The lifetime of the [local variable](#_Trm00193) must now be extended to at least the lifetime of the anonymous function delegate. This can be achieved by "hoisting" the [local variable](#_Trm00193) into a field of a compiler generated class. Instantiation of the [local variable](#_Trm00193) ([§7.15.5.2](#_Toc00328)) then corresponds to creating an [instance](#_Trm00172) of the compiler generated class, and accessing the [local variable](#_Trm00193) corresponds to accessing a field in the [instance](#_Trm00172) of the compiler generated class. Furthermore, the anonymous function becomes an [instance](#_Trm00172) [method](#_Trm00056) 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 variable](#_Trm00193)s 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](#_Trm00038) in which locals are captured such that the locals in the different [block](#_Trm00038)s can have independent lifetimes. An [instance](#_Trm00172) of \_\_Locals2, the compiler generated class for the inner statement [block](#_Trm00038), contains the [local variable](#_Trm00193) z and a field that [references](#_Trm00160) an [instance](#_Trm00172) of \_\_Locals1. An [instance](#_Trm00172) of \_\_Locals1, the compiler generated class for the outer statement [block](#_Trm00038), contains the [local variable](#_Trm00193) y and a field that [references](#_Trm00160) this of the enclosing function member. With these data structures it is possible to reach all captured outer [variables](#_Trm00031) through an [instance](#_Trm00172) of \_\_Local2, and the code of the anonymous function can thus be implemented as an [instance](#_Trm00172) [method](#_Trm00056) 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 variable](#_Trm00193)s can also be used when converting anonymous functions to expression trees: References to the compiler generated [object](#_Trm00173)s can be stored in the expression tree, and access to the [local variable](#_Trm00193)s can be represented as field accesses on these [object](#_Trm00173)s. The advantage of this approach is that it allows the "lifted" [local variable](#_Trm00193)s to be shared between delegates and expression trees.

## Method group [conversion](#_Trm00196)s

An [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) exists from a [method](#_Trm00056) group ([§7.1](#_Toc00205)) to a compatible [delegate type](#_Trm00107). Given a [delegate type](#_Trm00107) D and an expression E that is classified as a [method](#_Trm00056) group, an [implicit](#_Trm00197) [conversion](#_Trm00196) exists from E to D if E contains at least one [method](#_Trm00056) that is applicable in its normal form ([§7.5.3.1](#_Toc00243)) to an argument list constructed by use of the parameter [types](#_Trm00011) and modifiers of D, as described in the following.

The compile-time [application](#_Trm00124) of a [conversion](#_Trm00196) from a [method](#_Trm00056) group E to a [delegate type](#_Trm00107) D is described in the following. Note that the existence of an [implicit](#_Trm00197) [conversion](#_Trm00196) from E to D does not guarantee that the compile-time [application](#_Trm00124) of the [conversion](#_Trm00196) will succeed without error.

* A single [method](#_Trm00056) M is selected corresponding to a [method](#_Trm00056) invocation ([§7.6.5.1](#_Toc00261)) 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](#_Grm00117) of D.
  + The candidate [method](#_Trm00056)s considered are only those [method](#_Trm00056)s that are applicable in their normal form ([§7.5.3.1](#_Toc00243)), not those applicable only in their expanded form.
* If the algorithm of [§7.6.5.1](#_Toc00261) produces an error, then a compile-time error occurs. Otherwise the algorithm produces a single best [method](#_Trm00056) M having the same number of [parameters](#_Trm00059) as D and the [conversion](#_Trm00196) is considered to exist.
* The selected [method](#_Trm00056) M must be compatible ([§15.2](#_Toc00559)) with the [delegate type](#_Trm00107) D, or otherwise, a compile-time error occurs.
* If the selected [method](#_Trm00056) M is an [instance](#_Trm00172) [method](#_Trm00056), the [instance](#_Trm00172) expression associated with E determines the target [object](#_Trm00173) of the delegate.
* If the selected [method](#_Trm00056) M is an extension [method](#_Trm00056) which is denoted by means of a member access on an [instance](#_Trm00172) expression, that [instance](#_Trm00172) expression determines the target [object](#_Trm00173) of the delegate.
* The result of the [conversion](#_Trm00196) is a value of type D, namely a newly created delegate that refers to the selected [method](#_Trm00056) and target [object](#_Trm00173).
* Note that this process can lead to the creation of a delegate to an extension [method](#_Trm00056), if the algorithm of [§7.6.5.1](#_Toc00261) fails to find an [instance](#_Trm00172) [method](#_Trm00056) but succeeds in processing the invocation of E(A) as an extension [method](#_Trm00056) invocation ([§7.6.5.2](#_Toc00262)). A delegate thus created captures the extension [method](#_Trm00056) as well as its first argument.

The following example demonstrates [method](#_Trm00056) group [conversion](#_Trm00196)s:

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 [implicit](#_Trm00197)ly converts the [method](#_Trm00056) group F to a value of type D1.

The assignment to d2 shows how it is possible to create a delegate to a [method](#_Trm00056) that has less derived (contra-variant) parameter [types](#_Trm00011) and a more derived (covariant) [return type](#_Trm00060).

The assignment to d3 shows how no [conversion](#_Trm00196) exists if the [method](#_Trm00056) is not applicable.

The assignment to d4 shows how the [method](#_Trm00056) must be applicable in its normal form.

The assignment to d5 shows how parameter and [return type](#_Trm00060)s of the delegate and [method](#_Trm00056) are allowed to differ only for [reference types](#_Trm00019).

As with all other [implicit](#_Trm00197) and [explicit](#_Trm00198) [conversion](#_Trm00196)s, the cast [operator](#_Trm00090) can be used to [explicit](#_Trm00198)ly perform a [method](#_Trm00056) group [conversion](#_Trm00196). Thus, the example

object obj = new EventHandler(myDialog.OkClick);

could instead be written

object obj = (EventHandler)myDialog.OkClick;

Method groups may influence [overload resolution](#_Trm00078), and participate in type inference. See [§7.5](#_Toc00223) for further details.

The run-time evaluation of a [method](#_Trm00056) group [conversion](#_Trm00196) proceeds as follows:

* If the [method](#_Trm00056) selected at compile-time is an [instance](#_Trm00172) [method](#_Trm00056), or it is an extension [method](#_Trm00056) which is accessed as an [instance](#_Trm00172) [method](#_Trm00056), the target [object](#_Trm00173) of the delegate is determined from the [instance](#_Trm00172) expression associated with E:
  + The [instance](#_Trm00172) expression is evaluated. If this evaluation causes an exception, no further steps are executed.
  + If the [instance](#_Trm00172) expression is of a [reference\_type](#_Grm00030), the value computed by the [instance](#_Trm00172) expression becomes the target [object](#_Trm00173). If the selected [method](#_Trm00056) is an [instance](#_Trm00172) [method](#_Trm00056) and the target [object](#_Trm00173) is null, a System.NullReferenceException is thrown and no further steps are executed.
  + If the [instance](#_Trm00172) expression is of a [value\_type](#_Grm00029), a [boxing](#_Trm00029) operation ([§4.3.1](#_Toc00111)) is performed to convert the value to an [object](#_Trm00173), and this [object](#_Trm00173) becomes the target [object](#_Trm00173).
* Otherwise the selected [method](#_Trm00056) is part of a static [method](#_Trm00056) call, and the target [object](#_Trm00173) of the delegate is null.
* A new [instance](#_Trm00172) of the [delegate type](#_Trm00107) D is allocated. If there is not enough memory available to allocate the new [instance](#_Trm00172), a System.OutOfMemoryException is thrown and no further steps are executed.
* The new delegate [instance](#_Trm00172) is initialized with a reference to the [method](#_Trm00056) that was determined at compile-time and a reference to the target [object](#_Trm00173) computed above.

# [Expressions](#_Trm00032)

An expression is a sequence of [operator](#_Trm00090)s and [operands](#_Trm00033). This chapter defines the syntax, order of evaluation of [operands](#_Trm00033) and [operator](#_Trm00090)s, 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](#_Grm00038) ([§7.6.4](#_Toc00257)). 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](#_Grm00038) ([§7.6.4](#_Toc00257)), or as an operand for the as [operator](#_Trm00090) ([§7.10.11](#_Toc00310)), the is [operator](#_Trm00090) ([§7.10.10](#_Toc00309)), or the typeof [operator](#_Trm00090) ([§7.6.11](#_Toc00277)). In any other context, an expression classified as a type causes a compile-time error.
* A [method](#_Trm00056) group, which is a set of [overloaded](#_Trm00036) [method](#_Trm00056)s resulting from a member lookup ([§7.4](#_Toc00221)). A [method](#_Trm00056) group may have an associated [instance](#_Trm00172) expression and an associated type argument list. When an [instance](#_Trm00172) [method](#_Trm00056) is invoked, the result of evaluating the [instance](#_Trm00172) expression becomes the [instance](#_Trm00172) represented by this ([§7.6.7](#_Toc00267)). A [method](#_Trm00056) group is permitted in an [invocation\_expression](#_Grm00039) ([§7.6.5](#_Toc00260)) , a [delegate\_creation\_expression](#_Grm00048) ([§7.6.10.5](#_Toc00275)) and as the left hand side of an is [operator](#_Trm00090), and can be [implicit](#_Trm00197)ly converted to a compatible [delegate type](#_Trm00107) ([§6.6](#_Toc00203)). In any other context, an expression classified as a [method](#_Trm00056) group causes a compile-time error.
* A null [literal](#_Trm00118). An expression with this classification can be [implicit](#_Trm00197)ly converted to a reference type or nullable type.
* An anonymous function. An expression with this classification can be [implicit](#_Trm00197)ly converted to a compatible [delegate type](#_Trm00107) 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](#_Trm00172) expression. When an accessor (the get or set [block](#_Trm00038)) of an [instance](#_Trm00172) property access is invoked, the result of evaluating the [instance](#_Trm00172) expression becomes the [instance](#_Trm00172) represented by this ([§7.6.7](#_Toc00267)).
* An [event](#_Trm00088) access. Every [event](#_Trm00088) access has an associated type, namely the type of the [event](#_Trm00088). Furthermore, an [event](#_Trm00088) access may have an associated [instance](#_Trm00172) expression. An [event](#_Trm00088) access may appear as the left hand operand of the += and -= [operator](#_Trm00090)s ([§7.17.3](#_Toc00344)). In any other context, an expression classified as an [event](#_Trm00088) access causes a compile-time error.
* An [indexer](#_Trm00087) access. Every [indexer](#_Trm00087) access has an associated type, namely the [element type](#_Trm00095) of the [indexer](#_Trm00087). Furthermore, an [indexer](#_Trm00087) access has an associated [instance](#_Trm00172) expression and an associated argument list. When an accessor (the get or set [block](#_Trm00038)) of an [indexer](#_Trm00087) access is invoked, the result of evaluating the [instance](#_Trm00172) expression becomes the [instance](#_Trm00172) represented by this ([§7.6.7](#_Toc00267)), 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](#_Trm00056) with a [return type](#_Trm00060) of void. An expression classified as nothing is only valid in the context of a [statement\_expression](#_Grm00078) ([§8.6](#_Toc00357)).

The final result of an expression is never a namespace, type, [method](#_Trm00056) group, or [event](#_Trm00088) access. Rather, as noted above, these categories of expressions are intermediate con[structs](#_Trm00092) that are only permitted in certain contexts.

A property access or [indexer](#_Trm00087) 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](#_Trm00087) access: If the access is the target of an assignment, the *set accessor* is invoked to assign a new value ([§7.17.1](#_Toc00342)). Otherwise, the *get accessor* is invoked to obtain the current value ([§7.1.1](#_Toc00206)).

### Values of expressions

Most of the con[structs](#_Trm00092) 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](#_Trm00056) group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an [indexer](#_Trm00087) access, or a variable, the [value](#_Trm00209) of the property, [indexer](#_Trm00087), or variable is [implicit](#_Trm00197)ly substituted:

* The [value](#_Trm00209) of a variable is simply the [value](#_Trm00209) currently stored in the storage location identified by the variable. A variable must be considered [definitely assigned](#_Trm00068) ([§5.3](#_Toc00132)) before its [value](#_Trm00209) can be obtained, or otherwise a compile-time error occurs.
* The [value](#_Trm00209) of a property access expression is obtained by invoking the *get accessor* of the property. If the property has no *get accessor*, a compile-time error occurs. Otherwise, a function member invocation ([§7.5.4](#_Toc00249)) is performed, and the result of the invocation becomes the [value](#_Trm00209) of the property access expression.
* The [value](#_Trm00209) of an [indexer](#_Trm00087) access expression is obtained by invoking the *get accessor* of the [indexer](#_Trm00087). If the [indexer](#_Trm00087) has no *get accessor*, a compile-time error occurs. Otherwise, a function member invocation ([§7.5.4](#_Toc00249)) is performed with the argument list associated with the [indexer](#_Trm00087) access expression, and the result of the invocation becomes the [value](#_Trm00209) of the [indexer](#_Trm00087) access expression.

## Static and Dynamic Binding

The process of determining the meaning of an operation based on the type or [value](#_Trm00209) of constituent expressions ([arguments](#_Trm00062), [operands](#_Trm00033), receivers) is often referred to as ***binding***. For [instance](#_Trm00172) the meaning of a [method](#_Trm00056) call is determined based on the type of the receiver and [arguments](#_Trm00062). The meaning of an [operator](#_Trm00090) is determined based on the type of its [operands](#_Trm00033).

In C# the meaning of an operation is usually determined at compile-time, based on the [compile-time type](#_Trm00074) 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](#_Trm00174) (i.e. has the type dynamic) this indicates that any [binding](#_Trm00210) that it participates in should be based on its [run-time type](#_Trm00073) (i.e. the actual type of the [object](#_Trm00173) it denotes at run-time) rather than the type it has at compile-time. The [binding](#_Trm00210) of such an operation is therefore deferred until the time where the operation is to be executed during the running of the [program](#_Trm00109). 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](#_Trm00210) fails, errors are reported as exceptions at run-time.

The following operations in C# are subject to [binding](#_Trm00210):

* Member access: e.M
* Method invocation: e.M(e1, ..., eN)
* Delegate invocaton:e(e1, ..., eN)
* Element access: e[e1, ..., eN]
* Object creation: new C(e1, ..., eN)
* Overloaded unary [operator](#_Trm00090)s: +, -, !, ~, ++, --, true, false
* Overloaded binary [operator](#_Trm00090)s: +, -, \*, /, %, &, &&, |, ||, ??, ^, <<, >>, ==,!=, >, <, >=, <=
* Assignment [operator](#_Trm00090)s: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
* Implicit and [explicit](#_Trm00198) [conversion](#_Trm00196)s

When no [dynamic expression](#_Trm00174)s are involved, C# defaults to static [binding](#_Trm00210), which means that the [compile-time type](#_Trm00074)s 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](#_Trm00174), the operation is instead dynamically bound.

### Binding-time

Static [binding](#_Trm00210) takes place at compile-time, whereas dynamic [binding](#_Trm00210) 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](#_Trm00210) takes place.

The following example illustrates the notions of static and dynamic [binding](#_Trm00210) and of [binding](#_Trm00210)-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](#_Trm00074) of their argument. Thus, the [binding](#_Trm00210)-time is compile-time.

The third call is dynamically bound: the overload of Console.WriteLine is picked based on the [run-time type](#_Trm00073) of its argument. This happens because the argument is a [dynamic expression](#_Trm00174) -- its [compile-time type](#_Trm00074) is dynamic. Thus, the [binding](#_Trm00210)-time for the third call is run-time.

### Dynamic [binding](#_Trm00210)

The purpose of dynamic [binding](#_Trm00210) is to allow C# [program](#_Trm00109)s to interact with ***dynamic objects***, i.e. [object](#_Trm00173)s that do not follow the normal rules of the C# type system. Dynamic [object](#_Trm00173)s may be [object](#_Trm00173)s from other [program](#_Trm00109)ming languages with different [types](#_Trm00011) systems, or they may be [object](#_Trm00173)s that are [program](#_Trm00109)matically setup to implement their own [binding](#_Trm00210) semantics for different operations.

The mechanism by which a dynamic [object](#_Trm00173) implements its own semantics is implementation [defined](#_Trm00121). A given [interface](#_Trm00102) -- again implementation [defined](#_Trm00121) -- is implemented by [dynamic objects](#_Trm00214) to signal to the C# run-time that they have special semantics. Thus, whenever operations on a dynamic [object](#_Trm00173) are dynamically bound, their own [binding](#_Trm00210) semantics, rather than those of C# as specified in this document, take over.

While the purpose of dynamic [binding](#_Trm00210) is to allow interoperation with [dynamic objects](#_Trm00214), C# allows dynamic [binding](#_Trm00210) on all [object](#_Trm00173)s, whether they are dynamic or not. This allows for a smoother integration of [dynamic objects](#_Trm00214), as the results of operations on them may not themselves be [dynamic objects](#_Trm00214), but are still of a type unknown to the [program](#_Trm00109)mer at compile-time. Also dynamic [binding](#_Trm00210) can help eliminate error-prone reflection-based code even when no [object](#_Trm00173)s involved are [dynamic objects](#_Trm00214).

The following sections describe for each construct in the language exactly when dynamic [binding](#_Trm00210) 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](#_Trm00074) 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](#_Trm00074) of the constituent expression:

* A constituent expression of [compile-time type](#_Trm00074) dynamic is considered to have the type of the actual [value](#_Trm00209) that the expression evaluates to at runtime
* A constituent expression whose [compile-time type](#_Trm00074) 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](#_Trm00074).

## Operators

[Expressions](#_Trm00032) are constructed from ***operands*** and ***operators***. The [operator](#_Trm00090)s of an expression indicate which operations to apply to the [operands](#_Trm00033). Examples of [operator](#_Trm00090)s include +, -, \*, /, and new. Examples of [operands](#_Trm00033) include [literal](#_Trm00118)s, fields, [local variable](#_Trm00193)s, and expressions.

There are three kinds of [operator](#_Trm00090)s:

* Unary [operator](#_Trm00090)s. The unary [operator](#_Trm00090)s take one operand and use either prefix notation (such as --x) or postfix notation (such as x++).
* Binary [operator](#_Trm00090)s. The binary [operator](#_Trm00090)s take two [operands](#_Trm00033) and all use infix notation (such as x + y).
* Ternary [operator](#_Trm00090). Only one ternary [operator](#_Trm00090), ?:, exists; it takes three [operands](#_Trm00033) and uses infix notation (c ? x : y).

The order of evaluation of [operator](#_Trm00090)s in an expression is determined by the ***precedence*** and ***associativity*** of the [operator](#_Trm00090)s ([§7.3.1](#_Toc00212)).

Operands in an expression are evaluated from left to right. For example, in F(i) + G(i++) \* H(i), [method](#_Trm00056) F is called using the old [value](#_Trm00209) of i, then [method](#_Trm00056) G is called with the old [value](#_Trm00209) of i, and, finally, [method](#_Trm00056) H is called with the new [value](#_Trm00209) of i. This is separate from and unrelated to [operator](#_Trm00090) [precedence](#_Trm00035).

Certain [operator](#_Trm00090)s can be ***overloaded***. Operator [overloading](#_Trm00077) permits user-[defined](#_Trm00121) [operator](#_Trm00090) implementations to be specified for operations where one or both of the [operands](#_Trm00033) are of a user-[defined](#_Trm00121) class or struct type ([§7.3.2](#_Toc00213)).

### Operator [precedence](#_Trm00035) and [associativity](#_Trm00218)

When an expression contains multiple [operator](#_Trm00090)s, the ***precedence*** of the [operator](#_Trm00090)s controls the order in which the individual [operator](#_Trm00090)s are evaluated. For example, the expression x + y \* z is evaluated as x + (y \* z) because the \* [operator](#_Trm00090) has higher [precedence](#_Trm00035) than the binary + [operator](#_Trm00090). The [precedence](#_Trm00035) of an [operator](#_Trm00090) is established by the definition of its associated grammar production. For example, an [additive\_expression](#_Grm00057) consists of a sequence of [multiplicative\_expression](#_Grm00057)s separated by + or - [operator](#_Trm00090)s, thus giving the + and - [operator](#_Trm00090)s lower [precedence](#_Trm00035) than the \*, /, and % [operator](#_Trm00090)s.

The following table summarizes all [operator](#_Trm00090)s in order of [precedence](#_Trm00035) from highest to lowest:

|  |  |  |
| --- | --- | --- |
| **Section** | **Category** | **Operators** |
| [§7.6](#_Toc00252) | Primary | x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate |
| [§7.7](#_Toc00281) | Unary | + \* ! ~ ++x --x (T)x |
| [§7.8](#_Toc00292) | Multiplicative | \* / % |
| [§7.8](#_Toc00292) | Additive | + - |
| [§7.9](#_Toc00298) | Shift | << >> |
| [§7.10](#_Toc00299) | Relational and type testing | < > <= >= is as |
| [§7.10](#_Toc00299) | Equality | == != |
| [§7.11](#_Toc00311) | Logical AND | & |
| [§7.11](#_Toc00311) | Logical XOR | ^ |
| [§7.11](#_Toc00311) | Logical OR | | |
| [§7.12](#_Toc00316) | Conditional AND | && |
| [§7.12](#_Toc00316) | Conditional OR | || |
| [§7.13](#_Toc00319) | Null coalescing | ?? |
| [§7.14](#_Toc00320) | Conditional | ?: |
| [§7.17](#_Toc00341), [§7.15](#_Toc00321) | Assignment and lambda expression | = \*= /= %= += -= <<= >>= &= ^= |= => |

When an operand occurs between two [operator](#_Trm00090)s with the same [precedence](#_Trm00035), the [associativity](#_Trm00218) of the [operator](#_Trm00090)s controls the order in which the operations are performed:

* Except for the assignment [operator](#_Trm00090)s and the null coalescing [operator](#_Trm00090), all binary [operator](#_Trm00090)s 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 [operator](#_Trm00090)s, the null coalescing [operator](#_Trm00090) and the conditional [operator](#_Trm00090) (?:) 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](#_Trm00218) 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](#_Trm00077)

All unary and binary [operator](#_Trm00090)s have pre[defined](#_Trm00121) implementations that are automatically available in any expression. In addition to the pre[defined](#_Trm00121) implementations, user-[defined](#_Trm00121) implementations can be introduced by including operator declarations in classes and [structs](#_Trm00092) ([§10.10](#_Toc00470)). User-[defined](#_Trm00121) [operator](#_Trm00090) implementations always take [precedence](#_Trm00035) over pre[defined](#_Trm00121) [operator](#_Trm00090) implementations: Only when no applicable user-[defined](#_Trm00121) [operator](#_Trm00090) implementations exist will the pre[defined](#_Trm00121) [operator](#_Trm00090) implementations be considered, as described in [§7.3.3](#_Toc00214) and [§7.3.4](#_Toc00215).

The ***overloadable unary operators*** are:

+ - ! ~ ++ -- true false

Although true and false are not used [explicit](#_Trm00198)ly in expressions (and therefore are not included in the [precedence](#_Trm00035) table in [§7.3.1](#_Toc00212)), they are considered [operator](#_Trm00090)s because they are invoked in several expression contexts: boolean expressions ([§7.20](#_Toc00347)) and expressions involving the conditional ([§7.14](#_Toc00320)), and conditional logical [operator](#_Trm00090)s ([§7.12](#_Toc00316)).

The ***overloadable binary operators*** are:

+ - \* / % & | ^ << >> == != > < >= <=

Only the [operator](#_Trm00090)s listed above can be [overloaded](#_Trm00036). In particular, it is not possible to overload member access, [method](#_Trm00056) invocation, or the =, &&, ||, ??, ?:, =>, checked, unchecked, new, typeof, default, as, and is [operator](#_Trm00090)s.

When a binary [operator](#_Trm00090) is [overloaded](#_Trm00036), the corresponding assignment [operator](#_Trm00090), if any, is also [implicit](#_Trm00197)ly [overloaded](#_Trm00036). For example, an overload of [operator](#_Trm00090) \* is also an overload of [operator](#_Trm00090) \*=. This is described further in [§7.17.2](#_Toc00343). Note that the assignment [operator](#_Trm00090) itself (=) cannot be [overloaded](#_Trm00036). An assignment always performs a simple bit-wise copy of a [value](#_Trm00209) into a variable.

Cast operations, such as (T)x, are [overloaded](#_Trm00036) by providing user-[defined](#_Trm00121) [conversion](#_Trm00196)s ([§6.4](#_Toc00193)).

Element access, such as a[x], is not considered an overloadable [operator](#_Trm00090). Instead, user-[defined](#_Trm00121) indexing is supported through [indexer](#_Trm00087)s ([§10.9](#_Toc00468)).

In expressions, [operator](#_Trm00090)s are referenced using [operator](#_Trm00090) notation, and in declarations, [operator](#_Trm00090)s are referenced using functional notation. The following table shows the relationship between [operator](#_Trm00090) and functional notations for unary and binary [operator](#_Trm00090)s. In the first entry, *op* denotes any overloadable unary prefix [operator](#_Trm00090). In the second entry, *op* denotes the unary postfix ++ and -- [operator](#_Trm00090)s. In the third entry, *op* denotes any overloadable binary [operator](#_Trm00090).

|  |  |
| --- | --- |
| **Operator notation** | **Functional notation** |
| op x | operator op(x) |
| x op | operator op(x) |
| x op y | operator op(x,y) |

User-[defined](#_Trm00121) [operator](#_Trm00090) declarations always require at least one of the [parameters](#_Trm00059) to be of the class or struct type that contains the [operator](#_Trm00090) declaration. Thus, it is not possible for a user-[defined](#_Trm00121) [operator](#_Trm00090) to have the same [signature](#_Trm00061) as a pre[defined](#_Trm00121) [operator](#_Trm00090).

User-[defined](#_Trm00121) [operator](#_Trm00090) declarations cannot modify the syntax, [precedence](#_Trm00035), or [associativity](#_Trm00218) of an [operator](#_Trm00090). For example, the / [operator](#_Trm00090) is always a binary [operator](#_Trm00090), always has the [precedence](#_Trm00035) level specified in [§7.3.1](#_Toc00212), and is always [left-associative](#_Trm00221).

While it is possible for a user-[defined](#_Trm00121) [operator](#_Trm00090) 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](#_Trm00033) for equality and return an appropriate bool result.

The descriptions of individual [operator](#_Trm00090)s in [§7.6](#_Toc00252) through [§7.12](#_Toc00316) specify the pre[defined](#_Trm00121) implementations of the [operator](#_Trm00090)s and any additional rules that apply to each [operator](#_Trm00090). 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](#_Trm00090) [overload resolution](#_Trm00078)

An operation of the form op x or x op, where op is an overloadable unary [operator](#_Trm00090), and x is an expression of type X, is processed as follows:

* The set of candidate user-[defined](#_Trm00121) [operator](#_Trm00090)s provided by X for the operation operator op(x) is determined using the rules of [§7.3.5](#_Toc00216).
* If the set of candidate user-[defined](#_Trm00121) [operator](#_Trm00090)s is not empty, then this becomes the set of candidate [operator](#_Trm00090)s for the operation. Otherwise, the pre[defined](#_Trm00121) unary operator op implementations, including their lifted forms, become the set of candidate [operator](#_Trm00090)s for the operation. The pre[defined](#_Trm00121) implementations of a given [operator](#_Trm00090) are specified in the description of the [operator](#_Trm00090) ([§7.6](#_Toc00252) and [§7.7](#_Toc00281)).
* The [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242) are applied to the set of candidate [operator](#_Trm00090)s to select the best [operator](#_Trm00090) with respect to the argument list (x), and this [operator](#_Trm00090) becomes the result of the [overload resolution](#_Trm00078) process. If [overload resolution](#_Trm00078) fails to select a single best [operator](#_Trm00090), a [binding](#_Trm00210)-time error occurs.

### Binary [operator](#_Trm00090) [overload resolution](#_Trm00078)

An operation of the form x op y, where op is an overloadable binary [operator](#_Trm00090), 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](#_Trm00121) [operator](#_Trm00090)s provided by X and Y for the operation operator op(x,y) is determined. The set consists of the union of the candidate [operator](#_Trm00090)s provided by X and the candidate [operator](#_Trm00090)s provided by Y, each determined using the rules of [§7.3.5](#_Toc00216). If X and Y are the same type, or if X and Y are derived from a common base type, then shared candidate [operator](#_Trm00090)s only occur in the combined set once.
* If the set of candidate user-[defined](#_Trm00121) [operator](#_Trm00090)s is not empty, then this becomes the set of candidate [operator](#_Trm00090)s for the operation. Otherwise, the pre[defined](#_Trm00121) binary operator op implementations, including their lifted forms, become the set of candidate [operator](#_Trm00090)s for the operation. The pre[defined](#_Trm00121) implementations of a given [operator](#_Trm00090) are specified in the description of the [operator](#_Trm00090) ([§7.8](#_Toc00292) through [§7.12](#_Toc00316)). For pre[defined](#_Trm00121) enum and delegate [operator](#_Trm00090)s, the only [operator](#_Trm00090)s considered are those [defined](#_Trm00121) by an enum or [delegate type](#_Trm00107) that is the [binding](#_Trm00210)-time type of one of the [operands](#_Trm00033).
* The [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242) are applied to the set of candidate [operator](#_Trm00090)s to select the best [operator](#_Trm00090) with respect to the argument list (x,y), and this [operator](#_Trm00090) becomes the result of the [overload resolution](#_Trm00078) process. If [overload resolution](#_Trm00078) fails to select a single best [operator](#_Trm00090), a [binding](#_Trm00210)-time error occurs.

### Candidate user-[defined](#_Trm00121) [operator](#_Trm00090)s

Given a type T and an operation operator op(A), where op is an overloadable [operator](#_Trm00090) and A is an argument list, the set of candidate user-[defined](#_Trm00121) [operator](#_Trm00090)s 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](#_Trm00106), otherwise T0 is equal to T.
* For all operator op declarations in T0 and all lifted forms of such [operator](#_Trm00090)s, if at least one [operator](#_Trm00090) is applicable ([§7.5.3.1](#_Toc00243)) with respect to the argument list A, then the set of candidate [operator](#_Trm00090)s consists of all such applicable [operator](#_Trm00090)s in T0.
* Otherwise, if T0 is object, the set of candidate [operator](#_Trm00090)s is empty.
* Otherwise, the set of candidate [operator](#_Trm00090)s provided by T0 is the set of candidate [operator](#_Trm00090)s 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](#_Trm00197) [conversion](#_Trm00196)s of the [operands](#_Trm00033) of the pre[defined](#_Trm00121) unary and binary numeric [operator](#_Trm00090)s. Numeric promotion is not a distinct mechanism, but rather an effect of applying [overload resolution](#_Trm00078) to the pre[defined](#_Trm00121) [operator](#_Trm00090)s. Numeric promotion specifically does not affect evaluation of user-[defined](#_Trm00121) [operator](#_Trm00090)s, although user-[defined](#_Trm00121) [operator](#_Trm00090)s can be implemented to exhibit similar effects.

As an example of [numeric promotion](#_Trm00227), consider the pre[defined](#_Trm00121) implementations of the binary \* [operator](#_Trm00090):

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](#_Trm00078) rules ([§7.5.3](#_Toc00242)) are applied to this set of [operator](#_Trm00090)s, the effect is to select the first of the [operator](#_Trm00090)s for which [implicit](#_Trm00197) [conversion](#_Trm00196)s exist from the operand [types](#_Trm00011). For example, for the operation b \* s, where b is a byte and s is a short, [overload resolution](#_Trm00078) selects operator \*(int,int) as the best [operator](#_Trm00090). 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](#_Trm00078) selects operator \*(double,double) as the best [operator](#_Trm00090).

#### Unary [numeric promotion](#_Trm00227)s

Unary [numeric promotion](#_Trm00227) occurs for the [operands](#_Trm00033) of the pre[defined](#_Trm00121) +, -, and ~ unary [operator](#_Trm00090)s. Unary [numeric promotion](#_Trm00227) simply consists of converting [operands](#_Trm00033) of type sbyte, byte, short, ushort, or char to type int. Additionally, for the unary - [operator](#_Trm00090), unary [numeric promotion](#_Trm00227) converts [operands](#_Trm00033) of type uint to type long.

#### Binary [numeric promotion](#_Trm00227)s

Binary [numeric promotion](#_Trm00227) occurs for the [operands](#_Trm00033) of the pre[defined](#_Trm00121) +, -, \*, /, %, &, |, ^, ==, !=, >, <, >=, and <= binary [operator](#_Trm00090)s. Binary [numeric promotion](#_Trm00227) [implicit](#_Trm00197)ly converts both [operands](#_Trm00033) to a common type which, in case of the non-relational [operator](#_Trm00090)s, also becomes the result type of the operation. Binary [numeric promotion](#_Trm00227) 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](#_Trm00210)-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](#_Trm00210)-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](#_Trm00033) are converted to type long.
* Otherwise, if either operand is of type uint, the other operand is converted to type uint.
* Otherwise, both [operands](#_Trm00033) are converted to type int.

Note that the first rule disallows any operations that mix the decimal type with the double and float [types](#_Trm00011). The rule follows from the fact that there are no [implicit](#_Trm00197) [conversion](#_Trm00196)s between the decimal type and the double and float [types](#_Trm00011).

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](#_Trm00011).

In both of the above cases, a cast expression can be used to [explicit](#_Trm00198)ly 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](#_Trm00210)-time error occurs because a decimal cannot be multiplied by a double. The error is resolved by [explicit](#_Trm00198)ly converting the second operand to decimal, as follows:

decimal AddPercent(decimal x, double percent) {  
 return x \* (decimal)(1.0 + percent / 100.0);  
}

### Lifted [operator](#_Trm00090)s

***Lifted operators*** permit pre[defined](#_Trm00121) and user-[defined](#_Trm00121) [operator](#_Trm00090)s that operate on [non-nullable value type](#_Trm00169)s to also be used with nullable forms of those [types](#_Trm00011). [Lifted operators](#_Trm00228) are constructed from pre[defined](#_Trm00121) and user-[defined](#_Trm00121) [operator](#_Trm00090)s that meet certain requirements, as described in the following:

* For the unary [operator](#_Trm00090)s

+ ++ - -- ! ~

a lifted form of an [operator](#_Trm00090) exists if the operand and result [types](#_Trm00011) are both [non-nullable value type](#_Trm00169)s. The lifted form is constructed by adding a single ? modifier to the operand and result [types](#_Trm00011). The lifted [operator](#_Trm00090) produces a [null value](#_Trm00165) if the operand is null. Otherwise, the lifted [operator](#_Trm00090) unwraps the operand, applies the underlying [operator](#_Trm00090), and wraps the result.

* For the binary [operator](#_Trm00090)s

+ - \* / % & | ^ << >>

a lifted form of an [operator](#_Trm00090) exists if the operand and result [types](#_Trm00011) are all [non-nullable value type](#_Trm00169)s. The lifted form is constructed by adding a single ? modifier to each operand and result type. The lifted [operator](#_Trm00090) produces a [null value](#_Trm00165) if one or both [operands](#_Trm00033) are null (an exception being the & and | [operator](#_Trm00090)s of the bool? type, as described in [§7.11.3](#_Toc00314)). Otherwise, the lifted [operator](#_Trm00090) unwraps the [operands](#_Trm00033), applies the underlying [operator](#_Trm00090), and wraps the result.

* For the equality [operator](#_Trm00090)s

== !=

a lifted form of an [operator](#_Trm00090) exists if the operand [types](#_Trm00011) are both [non-nullable value type](#_Trm00169)s and if the result type is bool. The lifted form is constructed by adding a single ? modifier to each operand type. The lifted [operator](#_Trm00090) considers two [null value](#_Trm00165)s equal, and a [null value](#_Trm00165) unequal to any non-[null value](#_Trm00165). If both [operands](#_Trm00033) are non-null, the lifted [operator](#_Trm00090) unwraps the [operands](#_Trm00033) and applies the underlying [operator](#_Trm00090) to produce the bool result.

* For the relational [operator](#_Trm00090)s

< > <= >=

a lifted form of an [operator](#_Trm00090) exists if the operand [types](#_Trm00011) are both [non-nullable value type](#_Trm00169)s and if the result type is bool. The lifted form is constructed by adding a single ? modifier to each operand type. The lifted [operator](#_Trm00090) produces the [value](#_Trm00209) false if one or both [operands](#_Trm00033) are null. Otherwise, the lifted [operator](#_Trm00090) unwraps the [operands](#_Trm00033) and applies the underlying [operator](#_Trm00090) 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](#_Grm00036) ([§7.6.2](#_Toc00254)) or a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) in an expression. If the [simple\_name](#_Grm00036) or [member\_access](#_Grm00038) occurs as the [primary\_expression](#_Grm00035) of an [invocation\_expression](#_Grm00039) ([§7.6.5.1](#_Toc00261)), the member is said to be invoked.

If a member is a [method](#_Trm00056) or [event](#_Trm00088), or if it is a constant, field or property of either a [delegate type](#_Trm00107) ([§15](#_Toc00557)) or the type dynamic ([§4.2.3](#_Toc00105)), 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](#_Trm00059) the member has and whether the member is [accessible](#_Trm00138). For the purposes of member lookup, generic [method](#_Trm00056)s and [nested](#_Trm00143) [generic types](#_Trm00158) have the number of type [parameters](#_Trm00059) indicated in their respective declarations and all other [members](#_Trm00012) have zero type [parameters](#_Trm00059).

A member lookup of a name N with K type [parameters](#_Trm00059) in a type T is processed as follows:

* First, a set of [accessible](#_Trm00138) [members](#_Trm00012) named N is determined:
  + If T is a type parameter, then the set is the union of the sets of [accessible](#_Trm00138) [members](#_Trm00012) named N in each of the [types](#_Trm00011) specified as a primary constraint or secondary constraint ([§10.1.5](#_Toc00399)) for T, along with the set of [accessible](#_Trm00138) [members](#_Trm00012) named N in object.
  + Otherwise, the set consists of all [accessible](#_Trm00138) ([§3.5](#_Toc00076)) [members](#_Trm00012) named N in T, including [inherited](#_Trm00136) [members](#_Trm00012) and the [accessible](#_Trm00138) [members](#_Trm00012) named N in object. If T is a [constructed type](#_Trm00178), the set of [members](#_Trm00012) is obtained by substituting type [arguments](#_Trm00062) as described in [§10.3.2](#_Toc00412). Members that include an override modifier are excluded from the set.
* Next, if K is zero, all [nested](#_Trm00143) [types](#_Trm00011) whose declarations include type [parameters](#_Trm00059) are removed. If K is not zero, all [members](#_Trm00012) with a different number of type [parameters](#_Trm00059) are removed. Note that when K is zero, [method](#_Trm00056)s having type [parameters](#_Trm00059) are not removed, since the type inference process ([§7.5.2](#_Toc00227)) might be able to infer the type [arguments](#_Trm00062).
* Next, if the member is *invoked*, all non-*invocable* [members](#_Trm00012) are removed from the set.
* Next, [members](#_Trm00012) that are [hidden](#_Trm00150) by other [members](#_Trm00012) 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](#_Trm00088), or enumeration member, then all [members](#_Trm00012) declared in a base type of S are removed from the set.
  + If M is a type declaration, then all non-[types](#_Trm00011) declared in a base type of S are removed from the set, and all [type declarations](#_Trm00028) with the same number of type [parameters](#_Trm00059) as M declared in a base type of S are removed from the set.
  + If M is a [method](#_Trm00056), then all non-[method](#_Trm00056) [members](#_Trm00012) declared in a base type of S are removed from the set.
* Next, [interface](#_Trm00102) [members](#_Trm00012) that are [hidden](#_Trm00150) by class [members](#_Trm00012) 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](#_Trm00102) set ([§10.1.5](#_Toc00399)). 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](#_Trm00088), enumeration member, or type declaration, then all [members](#_Trm00012) declared in an [interface](#_Trm00102) declaration are removed from the set.
  + If M is a [method](#_Trm00056), then all non-[method](#_Trm00056) [members](#_Trm00012) declared in an [interface](#_Trm00102) declaration are removed from the set, and all [method](#_Trm00056)s with the same [signature](#_Trm00061) as M declared in an [interface](#_Trm00102) declaration are removed from the set.
* Finally, having removed [hidden](#_Trm00150) [members](#_Trm00012), the result of the lookup is determined:
  + If the set consists of a single member that is not a [method](#_Trm00056), then this member is the result of the lookup.
  + Otherwise, if the set contains only [method](#_Trm00056)s, then this group of [method](#_Trm00056)s is the result of the lookup.
  + Otherwise, the lookup is ambiguous, and a [binding](#_Trm00210)-time error occurs.

For member lookups in [types](#_Trm00011) other than type [parameters](#_Trm00059) and [interface](#_Trm00102)s, and member lookups in [interface](#_Trm00102)s that are strictly single-[inheritance](#_Trm00047) (each [interface](#_Trm00102) in the [inheritance](#_Trm00047) chain has exactly zero or one direct base [interface](#_Trm00102)), the effect of the lookup rules is simply that derived [members](#_Trm00012) [hide](#_Trm00132) base [members](#_Trm00012) with the same name or [signature](#_Trm00061). Such single-[inheritance](#_Trm00047) lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-[inheritance](#_Trm00047) [interface](#_Trm00102)s are described in [§13.2.5](#_Toc00541).

### Base [types](#_Trm00011)

For purposes of member lookup, a type T is considered to have the following base [types](#_Trm00011):

* If T is object, then T has no base type.
* If T is an [enum\_type](#_Grm00029), the base [types](#_Trm00011) of T are the [class types](#_Trm00024) System.Enum, System.ValueType, and object.
* If T is a [struct\_type](#_Grm00029), the base [types](#_Trm00011) of T are the [class types](#_Trm00024) System.ValueType and object.
* If T is a [class\_type](#_Grm00030), the base [types](#_Trm00011) of T are the [base classes](#_Trm00050) of T, including the class type object.
* If T is an [interface\_type](#_Grm00030), the base [types](#_Trm00011) of T are the base [interface](#_Trm00102)s of T and the class type object.
* If T is an [array\_type](#_Grm00030), the base [types](#_Trm00011) of T are the [class types](#_Trm00024) System.Array and object.
* If T is a [delegate\_type](#_Grm00030), the base [types](#_Trm00011) of T are the [class types](#_Trm00024) System.Delegate and object.

## Function [members](#_Trm00012)

Function [members](#_Trm00012) are [members](#_Trm00012) that contain executable [statements](#_Trm00037). Function [members](#_Trm00012) are always [members](#_Trm00012) of [types](#_Trm00011) and cannot be [members](#_Trm00012) of [namespaces](#_Trm00010). C# defines the following categories of [function members](#_Trm00079):

* Methods
* [Properties](#_Trm00082)
* Events
* Indexers
* User-[defined](#_Trm00121) [operator](#_Trm00090)s
* Instance constructors
* Static constructors
* Destructors

Except for [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s (which cannot be invoked [explicit](#_Trm00198)ly), the [statements](#_Trm00037) contained in [function members](#_Trm00079) are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list ([§7.5.1](#_Toc00224)) of a function member invocation provides actual [value](#_Trm00209)s or variable [references](#_Trm00160) for the [parameters](#_Trm00059) of the function member.

Invocations of generic [method](#_Trm00056)s may employ type inference to determine the set of type [arguments](#_Trm00062) to pass to the [method](#_Trm00056). This process is described in [§7.5.2](#_Toc00227).

Invocations of [method](#_Trm00056)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s and [instance](#_Trm00172) constructors employ [overload resolution](#_Trm00078) to determine which of a candidate set of [function members](#_Trm00079) to invoke. This process is described in [§7.5.3](#_Toc00242).

Once a particular function member has been identified at [binding](#_Trm00210)-time, possibly through [overload resolution](#_Trm00078), the actual run-time process of invoking the function member is described in [§7.5.4](#_Toc00249).

The following table summarizes the processing that takes place in con[structs](#_Trm00092) involving the six categories of [function members](#_Trm00079) that can be [explicit](#_Trm00198)ly invoked. In the table, e, x, y, and value indicate expressions classified as [variables](#_Trm00031) or [value](#_Trm00209)s, T indicates an expression classified as a type, F is the simple name of a [method](#_Trm00056), 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](#_Trm00056) F in the containing class or struct. The [method](#_Trm00056) is invoked with the argument list (x,y). If the [method](#_Trm00056) is not static, the [instance](#_Trm00172) expression is this. |
|  | T.F(x,y) | Overload resolution is applied to select the best [method](#_Trm00056) F in the class or struct T. A [binding](#_Trm00210)-time error occurs if the [method](#_Trm00056) is not static. The [method](#_Trm00056) is invoked with the argument list (x,y). |
|  | e.F(x,y) | Overload resolution is applied to select the best [method](#_Trm00056) F in the class, struct, or [interface](#_Trm00102) given by the type of e. A [binding](#_Trm00210)-time error occurs if the [method](#_Trm00056) is static. The [method](#_Trm00056) is invoked with the [instance](#_Trm00172) 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](#_Trm00172) 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](#_Trm00172) 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](#_Trm00102) given by the type of e is invoked with the [instance](#_Trm00172) expression e. A [binding](#_Trm00210)-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](#_Trm00102) given by the type of e is invoked with the [instance](#_Trm00172) expression e and the argument list (value). A [binding](#_Trm00210)-time error occurs if P is static or if P is read-only. |
| Event access | E += value | The add accessor of the [event](#_Trm00088) E in the containing class or struct is invoked. If E is not static, the [instance](#_Trm00172) expression is this. |
|  | E -= value | The remove accessor of the [event](#_Trm00088) E in the containing class or struct is invoked. If E is not static, the [instance](#_Trm00172) expression is this. |
|  | T.E += value | The add accessor of the [event](#_Trm00088) E in the class or struct T is invoked. A [binding](#_Trm00210)-time error occurs if E is not static. |
|  | T.E -= value | The remove accessor of the [event](#_Trm00088) E in the class or struct T is invoked. A [binding](#_Trm00210)-time error occurs if E is not static. |
|  | e.E += value | The add accessor of the [event](#_Trm00088) E in the class, struct, or [interface](#_Trm00102) given by the type of e is invoked with the [instance](#_Trm00172) expression e. A [binding](#_Trm00210)-time error occurs if E is static. |
|  | e.E -= value | The remove accessor of the [event](#_Trm00088) E in the class, struct, or [interface](#_Trm00102) given by the type of e is invoked with the [instance](#_Trm00172) expression e. A [binding](#_Trm00210)-time error occurs if E is static. |
| Indexer access | e[x,y] | Overload resolution is applied to select the best [indexer](#_Trm00087) in the class, struct, or [interface](#_Trm00102) given by the type of e. The get accessor of the [indexer](#_Trm00087) is invoked with the [instance](#_Trm00172) expression e and the argument list (x,y). A [binding](#_Trm00210)-time error occurs if the [indexer](#_Trm00087) is write-only. |
|  | e[x,y] = value | Overload resolution is applied to select the best [indexer](#_Trm00087) in the class, struct, or [interface](#_Trm00102) given by the type of e. The set accessor of the [indexer](#_Trm00087) is invoked with the [instance](#_Trm00172) expression e and the argument list (x,y,value). A [binding](#_Trm00210)-time error occurs if the [indexer](#_Trm00087) is read-only. |
| Operator invocation | -x | Overload resolution is applied to select the best unary [operator](#_Trm00090) in the class or struct given by the type of x. The selected [operator](#_Trm00090) is invoked with the argument list (x). |
|  | x + y | Overload resolution is applied to select the best binary [operator](#_Trm00090) in the classes or [structs](#_Trm00092) given by the [types](#_Trm00011) of x and y. The selected [operator](#_Trm00090) is invoked with the argument list (x,y). |
| Instance constructor invocation | new T(x,y) | Overload resolution is applied to select the best [instance](#_Trm00172) constructor in the class or struct T. The [instance](#_Trm00172) constructor is invoked with the argument list (x,y). |

### Argument lists

Every function member and delegate invocation includes an argument list which provides actual [value](#_Trm00209)s or variable [references](#_Trm00160) for the [parameters](#_Trm00059) of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

* For [instance](#_Trm00172) constructors, [method](#_Trm00056)s, [indexer](#_Trm00087)s and delegates, the [arguments](#_Trm00062) are specified as an [argument\_list](#_Grm00034), as described below. For [indexer](#_Trm00087)s, when invoking the set accessor, the argument list additionally includes the expression specified as the right operand of the assignment [operator](#_Trm00090).
* 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](#_Trm00090) when invoking the set accessor.
* For [event](#_Trm00088)s, the argument list consists of the expression specified as the right operand of the += or -= [operator](#_Trm00090).
* For user-[defined](#_Trm00121) [operator](#_Trm00090)s, the argument list consists of the single operand of the unary [operator](#_Trm00090) or the two [operands](#_Trm00033) of the binary [operator](#_Trm00090).

The [arguments](#_Trm00062) of properties ([§10.7](#_Toc00457)), [event](#_Trm00088)s ([§10.8](#_Toc00463)), and user-[defined](#_Trm00121) [operator](#_Trm00090)s ([§10.10](#_Toc00470)) are always passed as [value](#_Trm00209) [parameters](#_Trm00059) ([§10.6.1.1](#_Toc00443)). The [arguments](#_Trm00062) of [indexer](#_Trm00087)s ([§10.9](#_Toc00468)) are always passed as [value](#_Trm00209) [parameters](#_Trm00059) ([§10.6.1.1](#_Toc00443)) or parameter [array](#_Trm00093)s ([§10.6.1.4](#_Toc00446)). Reference and [output parameter](#_Trm00065)s are not supported for these categories of [function members](#_Trm00079).

The [arguments](#_Trm00062) of an [instance](#_Trm00172) constructor, [method](#_Trm00056), [indexer](#_Trm00087) or delegate invocation are specified as an [argument\_list](#_Grm00034):

argument\_list:  
 | argument ( ',' argument )\*  
 ;  
  
argument:  
 | argument\_name? argument\_value  
 ;  
  
argument\_name:  
 | identifier ':'  
 ;  
  
argument\_value:  
 | expression  
 | 'ref' variable\_reference  
 | 'out' variable\_reference  
 ;

An [argument\_list](#_Grm00034) consists of one or more [argument](#_Grm00034)s, separated by commas. Each argument consists of an optional [argument\_name](#_Grm00034) followed by an [argument\_value](#_Grm00034). An [argument](#_Grm00034) with an [argument\_name](#_Grm00034) is referred to as a ***named argument***, whereas an [argument](#_Grm00034) without an [argument\_name](#_Grm00034) is a ***positional argument***. It is an error for a [positional argument](#_Trm00230) to appear after a [named argument](#_Trm00229) in an [argument\_list](#_Grm00034).

The [argument\_value](#_Grm00034) can take one of the following forms:

* An [expression](#_Grm00067), indicating that the argument is passed as a [value](#_Trm00209) parameter ([§10.6.1.1](#_Toc00443)).
* The [keyword](#_Trm00117) ref followed by a [variable\_reference](#_Grm00033) ([§5.4](#_Toc00165)), indicating that the argument is passed as a [reference parameter](#_Trm00064) ([§10.6.1.2](#_Toc00444)). A variable must be [definitely assigned](#_Trm00068) ([§5.3](#_Toc00132)) before it can be passed as a [reference parameter](#_Trm00064). The [keyword](#_Trm00117) out followed by a [variable\_reference](#_Grm00033) ([§5.4](#_Toc00165)), indicating that the argument is passed as an [output parameter](#_Trm00065) ([§10.6.1.3](#_Toc00445)). A variable is considered [definitely assigned](#_Trm00068) ([§5.3](#_Toc00132)) following a function member invocation in which the variable is passed as an [output parameter](#_Trm00065).

#### Corresponding [parameters](#_Trm00059)

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 [method](#_Trm00056)s and [indexer](#_Trm00087)s [defined](#_Trm00121) in classes, the parameter list is picked from the [most specific](#_Trm00204) declaration or override of the function member, starting with the static type of the receiver, and searching through its [base classes](#_Trm00050).
* For [interface](#_Trm00102) [method](#_Trm00056)s and [indexer](#_Trm00087)s, the parameter list is picked form the [most specific](#_Trm00204) definition of the member, starting with the [interface](#_Trm00102) type and searching through the base [interface](#_Trm00102)s. If no unique parameter list is found, a parameter list with in[accessible](#_Trm00138) names and no optional [parameters](#_Trm00059) is constructed, so that invocations cannot use named [parameters](#_Trm00059) or omit optional [arguments](#_Trm00062).
* For partial [method](#_Trm00056)s, the parameter list of the defining partial [method](#_Trm00056) declaration is used.
* For all other [function members](#_Trm00079) and delegates there is only a single parameter list, which is the one used.

The position of an argument or parameter is [defined](#_Trm00121) as the number of [arguments](#_Trm00062) or [parameters](#_Trm00059) preceding it in the argument list or parameter list.

The corresponding [parameters](#_Trm00059) for function member [arguments](#_Trm00062) are established as follows:

* Arguments in the [argument\_list](#_Grm00034) of [instance](#_Trm00172) constructors, [method](#_Trm00056)s, [indexer](#_Trm00087)s and delegates:
  + A [positional argument](#_Trm00230) where a fixed parameter occurs at the same position in the parameter list corresponds to that parameter.
  + A [positional argument](#_Trm00230) of a function member with a parameter [array](#_Trm00093) invoked in its normal form corresponds to the parameter [array](#_Trm00093), which must occur at the same position in the parameter list.
  + A [positional argument](#_Trm00230) of a function member with a parameter [array](#_Trm00093) 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](#_Trm00093).
  + A [named argument](#_Trm00229) corresponds to the parameter of the same name in the parameter list.
  + For [indexer](#_Trm00087)s, when invoking the set accessor, the expression specified as the right operand of the assignment [operator](#_Trm00090) corresponds to the [implicit](#_Trm00197) value parameter of the set accessor declaration.
* For properties, when invoking the get accessor there are no [arguments](#_Trm00062). When invoking the set accessor, the expression specified as the right operand of the assignment [operator](#_Trm00090) corresponds to the [implicit](#_Trm00197) value parameter of the set accessor declaration.
* For user-[defined](#_Trm00121) unary [operator](#_Trm00090)s (including [conversion](#_Trm00196)s), the single operand corresponds to the single parameter of the [operator](#_Trm00090) declaration.
* For user-[defined](#_Trm00121) binary [operator](#_Trm00090)s, the left operand corresponds to the first parameter, and the right operand corresponds to the second parameter of the [operator](#_Trm00090) declaration.

#### Run-time evaluation of argument lists

During the run-time processing of a function member invocation ([§7.5.4](#_Toc00249)), the expressions or variable [references](#_Trm00160) of an argument list are evaluated in order, from left to right, as follows:

* For a [value](#_Trm00209) parameter, the argument expression is evaluated and an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) to the corresponding parameter type is performed. The resulting [value](#_Trm00209) becomes the initial [value](#_Trm00209) of the [value](#_Trm00209) parameter in the function member invocation.
* For a reference or [output parameter](#_Trm00065), 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](#_Trm00065) is an [array](#_Trm00093) element of a [reference\_type](#_Grm00030), a run-time check is performed to ensure that the [element type](#_Trm00095) of the [array](#_Trm00093) is identical to the type of the parameter. If this check fails, a System.ArrayTypeMismatchException is thrown.

Methods, [indexer](#_Trm00087)s, and [instance](#_Trm00172) constructors may declare their right-most parameter to be a parameter [array](#_Trm00093) ([§10.6.1.4](#_Toc00446)). Such [function members](#_Trm00079) are invoked either in their normal form or in their expanded form depending on which is applicable ([§7.5.3.1](#_Toc00243)):

* When a function member with a parameter [array](#_Trm00093) is invoked in its normal form, the argument given for the parameter [array](#_Trm00093) must be a single expression that is [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the parameter [array](#_Trm00093) type. In this case, the parameter [array](#_Trm00093) acts precisely like a [value](#_Trm00209) parameter.
* When a function member with a parameter [array](#_Trm00093) is invoked in its expanded form, the invocation must specify zero or more positional [arguments](#_Trm00062) for the parameter [array](#_Trm00093), where each argument is an expression that is [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the [element type](#_Trm00095) of the parameter [array](#_Trm00093). In this case, the invocation creates an [instance](#_Trm00172) of the parameter [array](#_Trm00093) type with a [length](#_Trm00096) corresponding to the number of [arguments](#_Trm00062), initializes the [elements](#_Trm00094) of the [array](#_Trm00093) [instance](#_Trm00172) with the given argument [value](#_Trm00209)s, and uses the newly created [array](#_Trm00093) [instance](#_Trm00172) 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](#_Trm00093) co-variance rules ([§12.5](#_Toc00525)) permit a [value](#_Trm00209) of an [array](#_Trm00093) type A[] to be a reference to an [instance](#_Trm00172) of an [array](#_Trm00093) type B[], provided an [implicit](#_Trm00197) reference [conversion](#_Trm00196) exists from B to A. Because of these rules, when an [array](#_Trm00093) element of a [reference\_type](#_Grm00030) is passed as a reference or [output parameter](#_Trm00065), a run-time check is required to ensure that the actual [element type](#_Trm00095) of the [array](#_Trm00093) 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](#_Trm00095) of b is string and not object.

When a function member with a parameter [array](#_Trm00093) is invoked in its expanded form, the invocation is processed exactly as if an [array](#_Trm00093) creation expression with an [array](#_Trm00093) initializer ([§7.6.10.4](#_Toc00274)) was inserted around the expanded [parameters](#_Trm00059). For example, given the declaration

void F(int x, int y, params object[] args);

the following invocations of the expanded form of the [method](#_Trm00056)

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](#_Trm00093) is created when there are zero [arguments](#_Trm00062) given for the parameter [array](#_Trm00093).

When [arguments](#_Trm00062) are omitted from a function member with corresponding optional [parameters](#_Trm00059), the default [arguments](#_Trm00062) of the function member declaration are [implicit](#_Trm00197)ly passed. Because these are always constant, their evaluation will not impact the evaluation order of the remaining [arguments](#_Trm00062).

### Type inference

When a generic [method](#_Trm00056) is called without specifying type [arguments](#_Trm00062), a ***type inference*** process attempts to infer type [arguments](#_Trm00062) for the call. The presence of [type inference](#_Trm00231) allows a more convenient syntax to be used for calling a generic [method](#_Trm00056), and allows the [program](#_Trm00109)mer to avoid specifying redundant type information. For example, given the [method](#_Trm00056) 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](#_Trm00056) without [explicit](#_Trm00198)ly 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](#_Trm00231), the type [arguments](#_Trm00062) int and string are determined from the [arguments](#_Trm00062) to the [method](#_Trm00056).

Type inference occurs as part of the [binding](#_Trm00210)-time processing of a [method](#_Trm00056) invocation ([§7.6.5.1](#_Toc00261)) and takes place before the [overload resolution](#_Trm00078) step of the invocation. When a particular [method](#_Trm00056) group is specified in a [method](#_Trm00056) invocation, and no type [arguments](#_Trm00062) are specified as part of the [method](#_Trm00056) invocation, [type inference](#_Trm00231) is applied to each generic [method](#_Trm00056) in the [method](#_Trm00056) group. If [type inference](#_Trm00231) succeeds, then the inferred type [arguments](#_Trm00062) are used to determine the [types](#_Trm00011) of [arguments](#_Trm00062) for subsequent [overload resolution](#_Trm00078). If [overload resolution](#_Trm00078) chooses a generic [method](#_Trm00056) as the one to invoke, then the inferred type [arguments](#_Trm00062) are used as the actual type [arguments](#_Trm00062) for the invocation. If [type inference](#_Trm00231) for a particular [method](#_Trm00056) fails, that [method](#_Trm00056) does not participate in [overload resolution](#_Trm00078). The failure of [type inference](#_Trm00231), in and of itself, does not cause a [binding](#_Trm00210)-time error. However, it often leads to a [binding](#_Trm00210)-time error when [overload resolution](#_Trm00078) then fails to find any applicable [method](#_Trm00056)s.

If the supplied number of [arguments](#_Trm00062) is different than the number of [parameters](#_Trm00059) in the [method](#_Trm00056), then inference immediately fails. Otherwise, assume that the generic [method](#_Trm00056) has the following [signature](#_Trm00061):

Tr M<X1,...,Xn>(T1 x1, ..., Tm xm)

With a [method](#_Trm00056) call of the form M(E1...Em) the task of [type inference](#_Trm00231) is to find unique type [arguments](#_Trm00062) S1...Sn for each of the type [parameters](#_Trm00059) 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](#_Trm00062) for more type [variables](#_Trm00031) based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the second phase fixes type [variables](#_Trm00031) to specific [types](#_Trm00011) 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](#_Trm00056) is called. Type inference for [conversion](#_Trm00196) of [method](#_Trm00056) groups is described in [§7.5.2.13](#_Toc00240) and finding the best common type of a set of expressions is described in [§7.5.2.14](#_Toc00241).

#### The first phase

For each of the [method](#_Trm00056) [arguments](#_Trm00062) Ei:

* If Ei is an anonymous function, an *explicit parameter type inference* ([§7.5.2.7](#_Toc00234)) is made from Ei to Ti
* Otherwise, if Ei has a type U and xi is a [value](#_Trm00209) 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](#_Trm00031) Xi which do not *depend on* ([§7.5.2.5](#_Toc00232)) any Xj are fixed ([§7.5.2.11](#_Toc00238)).
* If no such type [variables](#_Trm00031) exist, all *unfixed* type [variables](#_Trm00031) 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](#_Trm00031) exist and there are still *unfixed* type [variables](#_Trm00031), [type inference](#_Trm00231) fails.
* Otherwise, if no further *unfixed* type [variables](#_Trm00031) exist, [type inference](#_Trm00231) succeeds.
* Otherwise, for all [arguments](#_Trm00062) Ei with corresponding parameter type Ti where the *output types* ([§7.5.2.4](#_Toc00231)) contain *unfixed* type [variables](#_Trm00031) Xj but the *input types* ([§7.5.2.3](#_Toc00230)) do not, an *output type inference* ([§7.5.2.6](#_Toc00233)) is made *from* Ei *to* Ti. Then the second phase is repeated.

#### Input [types](#_Trm00011)

If E is a [method](#_Trm00056) group or [implicit](#_Trm00197)ly typed anonymous function and T is a [delegate type](#_Trm00107) or expression tree type then all the parameter [types](#_Trm00011) of T are *input types* of E *with type* T.

#### Output [types](#_Trm00011)

If E is a [method](#_Trm00056) group or an anonymous function and T is a [delegate type](#_Trm00107) or expression tree type then the [return type](#_Trm00060) 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 inference](#_Trm00231)s

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](#_Trm00060) U ([§7.5.2.12](#_Toc00239)) and T is a [delegate type](#_Trm00107) or expression tree type with [return type](#_Trm00060) Tb, then a *lower-bound inference* ([§7.5.2.9](#_Toc00236)) is made *from* U *to* Tb.
* Otherwise, if E is a [method](#_Trm00056) group and T is a [delegate type](#_Trm00107) or expression tree type with parameter [types](#_Trm00011) T1...Tk and [return type](#_Trm00060) Tb, and [overload resolution](#_Trm00078) of E with the [types](#_Trm00011) T1...Tk yields a single [method](#_Trm00056) with [return type](#_Trm00060) 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 inference](#_Trm00231)s

An *explicit parameter type inference* is made *from* an expression E *to* a type T in the following way:

* If E is an [explicit](#_Trm00198)ly typed anonymous function with parameter [types](#_Trm00011) U1...Uk and T is a [delegate type](#_Trm00107) or expression tree type with parameter [types](#_Trm00011) V1...Vk then for each Ui an *exact inference* ([§7.5.2.8](#_Toc00235)) 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](#_Trm00093) type V1[...] and U is an [array](#_Trm00093) type U1[...] of the same [rank](#_Trm00099)
  + V is the type V1? and U is the type U1?
  + V is a [constructed type](#_Trm00178) C<V1...Vk>and U is a [constructed type](#_Trm00178) 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](#_Trm00093) type V1[...] and U is an [array](#_Trm00093) type U1[...] (or a type parameter whose effective base type is U1[...]) of the same [rank](#_Trm00099)
  + V is one of IEnumerable<V1>, ICollection<V1> or IList<V1> and U is a one-dimensional [array](#_Trm00093) type U1[](or a type parameter whose effective base type is U1[])
  + V is a constructed class, struct, [interface](#_Trm00102) or [delegate type](#_Trm00107) 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](#_Trm00102) 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](#_Trm00102) 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](#_Trm00093) 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](#_Trm00093) type U1[...] and V is an [array](#_Trm00093) type V1[...] of the same [rank](#_Trm00099)
  + U is one of IEnumerable<Ue>, ICollection<Ue> or IList<Ue> and V is a one-dimensional [array](#_Trm00093) type Ve[]
  + U is the type U1? and V is the type V1?
  + U is constructed class, struct, [interface](#_Trm00102) or [delegate type](#_Trm00107) C<U1...Uk> and V is a class, struct, [interface](#_Trm00102) or [delegate type](#_Trm00107) 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](#_Trm00093) 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](#_Trm00011) 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](#_Trm00011) Uj which are not identical to U are removed from the candidate set. For each lower bound U of Xi all [types](#_Trm00011) Uj to which there is *not* an [implicit](#_Trm00197) [conversion](#_Trm00196) from U are removed from the candidate set. For each upper bound U of Xi all [types](#_Trm00011) Uj from which there is *not* an [implicit](#_Trm00197) [conversion](#_Trm00196) to U are removed from the candidate set.
* If among the remaining candidate [types](#_Trm00011) Uj there is a unique type V from which there is an [implicit](#_Trm00197) [conversion](#_Trm00196) to all the other candidate [types](#_Trm00011), then Xi is fixed to V.
* Otherwise, [type inference](#_Trm00231) fails.

#### Inferred [return type](#_Trm00060)

The inferred [return type](#_Trm00060) of an anonymous function F is used during [type inference](#_Trm00231) and [overload resolution](#_Trm00078). The inferred [return type](#_Trm00060) can only be determined for an anonymous function where all parameter [types](#_Trm00011) are known, either because they are [explicit](#_Trm00198)ly given, provided through an anonymous function [conversion](#_Trm00196) or inferred during [type inference](#_Trm00231) on an enclosing generic [method](#_Trm00056) invocation.

The ***inferred result type*** is determined as follows:

* If the body of F is an [expression](#_Grm00067) that has a type, then the [inferred result type](#_Trm00232) of F is the type of that expression.
* If the body of F is a [block](#_Grm00071) and the set of expressions in the [block](#_Trm00038)'s return [statements](#_Trm00037) has a best common type T ([§7.5.2.14](#_Toc00241)), then the [inferred result type](#_Trm00232) of F is T.
* Otherwise, a result type cannot be inferred for F.

The ***inferred return type*** is determined as follows:

* If F is async and the body of F is either an expression classified as nothing ([§7.1](#_Toc00205)), or a statement [block](#_Trm00038) where no return [statements](#_Trm00037) have expressions, the [inferred return type](#_Trm00233) is System.Threading.Tasks.Task
* If F is async and has an [inferred result type](#_Trm00232) T, the [inferred return type](#_Trm00233) is System.Threading.Tasks.Task<T>.
* If F is non-async and has an [inferred result type](#_Trm00232) T, the [inferred return type](#_Trm00233) is T.
* Otherwise a [return type](#_Trm00060) cannot be inferred for F.

As an example of [type inference](#_Trm00231) involving anonymous functions, consider the Select extension [method](#_Trm00056) 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](#_Trm00056) 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](#_Trm00056) invocation ([§7.6.5.2](#_Toc00262)) of Select is processed by rewriting the invocation to a static [method](#_Trm00056) invocation:

IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);

Since type [arguments](#_Trm00062) were not [explicit](#_Trm00198)ly specified, [type inference](#_Trm00231) is used to infer the type [arguments](#_Trm00062). First, the customers argument is related to the source parameter, inferring T to be Customer. Then, using the anonymous function [type inference](#_Trm00231) process described above, c is given type Customer, and the expression c.Name is related to the [return type](#_Trm00060) 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](#_Trm00231) allows type information to "flow" between [arguments](#_Trm00062) in a generic [method](#_Trm00056) invocation. Given the [method](#_Trm00056):

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](#_Trm00060) 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](#_Trm00060) of f2, inferring Z to be double. Thus, the result of the invocation is of type double.

#### Type inference for [conversion](#_Trm00196) of [method](#_Trm00056) groups

Similar to calls of generic [method](#_Trm00056)s, [type inference](#_Trm00231) must also be applied when a [method](#_Trm00056) group M containing a generic [method](#_Trm00056) is converted to a given [delegate type](#_Trm00107) D ([§6.6](#_Toc00203)). Given a [method](#_Trm00056)

Tr M<X1...Xn>(T1 x1 ... Tm xm)

and the [method](#_Trm00056) group M being assigned to the [delegate type](#_Trm00107) D the task of [type inference](#_Trm00231) is to find type [arguments](#_Trm00062) S1...Sn so that the expression:

M<S1...Sn>

becomes compatible ([§15.1](#_Toc00558)) with D.

Unlike the [type inference](#_Trm00231) algorithm for generic [method](#_Trm00056) 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](#_Trm00231) fails. Otherwise, all Xi are *fixed* to corresponding Si, which are the result of [type inference](#_Trm00231).

#### 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 type](#_Trm00095)s of [implicit](#_Trm00197)ly typed [array](#_Trm00093)s and the [return type](#_Trm00060)s of anonymous functions with [block](#_Grm00071) bodies are found in this way.

Intuitively, given a set of expressions E1...Em this inference should be equivalent to calling a [method](#_Trm00056)

Tr M<X>(X x1 ... X xm)

with the Ei as [arguments](#_Trm00062).

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](#_Trm00210)-time mechanism for selecting the best function member to invoke given an argument list and a set of candidate [function members](#_Trm00079). Overload resolution selects the function member to invoke in the following distinct contexts within C#:

* Invocation of a [method](#_Trm00056) named in an [invocation\_expression](#_Grm00039) ([§7.6.5.1](#_Toc00261)).
* Invocation of an [instance](#_Trm00172) constructor named in an [object\_creation\_expression](#_Grm00044) ([§7.6.10.1](#_Toc00271)).
* Invocation of an [indexer](#_Trm00087) accessor through an [element\_access](#_Grm00040) ([§7.6.6](#_Toc00264)).
* Invocation of a pre[defined](#_Trm00121) or user-[defined](#_Trm00121) [operator](#_Trm00090) referenced in an expression ([§7.3.3](#_Toc00214) and [§7.3.4](#_Toc00215)).

Each of these contexts defines the set of candidate [function members](#_Trm00079) and the list of [arguments](#_Trm00062) in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a [method](#_Trm00056) invocation does not include [method](#_Trm00056)s marked override ([§7.4](#_Toc00221)), and [method](#_Trm00056)s in a base class are not candidates if any [method](#_Trm00056) in a derived class is applicable ([§7.6.5.1](#_Toc00261)).

Once the candidate [function members](#_Trm00079) 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](#_Trm00079), 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](#_Trm00079) with respect to the given argument list, provided that each function member is compared to all other [function members](#_Trm00079) using the rules in [§7.5.3.2](#_Toc00244). If there is not exactly one function member that is better than all other [function members](#_Trm00079), then the function member invocation is ambiguous and a [binding](#_Trm00210)-time error occurs.

The following sections define the exact meanings of the terms ***applicable function member*** and ***better function member***.

#### Applicable function member

A function member is said to be an ***applicable function member*** with respect to an argument list A when all of the following are true:

* Each argument in A corresponds to a parameter in the function member declaration as described in [§7.5.1.1](#_Toc00225), 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](#_Trm00209), ref, or out) is identical to the parameter passing mode of the corresponding parameter, and
  + for a [value](#_Trm00209) parameter or a parameter [array](#_Trm00093), an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) 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](#_Trm00093), 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](#_Trm00093) is not applicable in its [normal form](#_Trm00237), the function member may instead be applicable in its ***expanded form***:

* The [expanded form](#_Trm00238) is constructed by replacing the parameter [array](#_Trm00093) in the function member declaration with zero or more [value](#_Trm00209) [parameters](#_Trm00059) of the [element type](#_Trm00095) of the parameter [array](#_Trm00093) such that the number of [arguments](#_Trm00062) in the argument list A matches the total number of [parameters](#_Trm00059). If A has fewer [arguments](#_Trm00062) than the number of fixed [parameters](#_Trm00059) in the function member declaration, the [expanded form](#_Trm00238) of the function member cannot be constructed and is thus not applicable.
* Otherwise, the [expanded form](#_Trm00238) 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](#_Trm00209) parameter or a [value](#_Trm00209) parameter created by the expansion, an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) 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](#_Trm00235), 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](#_Trm00079) are constructed in the following way:

* The [expanded form](#_Trm00238) is used if the function member was applicable only in the [expanded form](#_Trm00238).
* Optional [parameters](#_Trm00059) with no corresponding [arguments](#_Trm00062) are removed from the parameter list
* The [parameters](#_Trm00059) 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 member](#_Trm00234)s Mp and Mq with parameter [types](#_Trm00011) {P1, P2, ..., Pn} and {Q1, Q2, ..., Qn}, Mp is [defined](#_Trm00121) to be a ***better function member*** than Mq if

* for each argument, the [implicit](#_Trm00197) [conversion](#_Trm00196) from Ex to Qx is not better than the [implicit](#_Trm00197) [conversion](#_Trm00196) from Ex to Px, and
* for at least one argument, the [conversion](#_Trm00196) from Ex to Px is better than the [conversion](#_Trm00196) from Ex to Qx.

When performing this evaluation, if Mp or Mq is applicable in its [expanded form](#_Trm00238), then Px or Qx refers to a parameter in the [expanded form](#_Trm00238) 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](#_Trm00196) to the corresponding Qi), the following tie-breaking rules are applied, in order, to determine the [better function member](#_Trm00235).

* If Mp is a non-generic [method](#_Trm00056) and Mq is a generic [method](#_Trm00056), then Mp is better than Mq.
* Otherwise, if Mp is applicable in its [normal form](#_Trm00237) and Mq has a params [array](#_Trm00093) and is applicable only in its [expanded form](#_Trm00238), then Mp is better than Mq.
* Otherwise, if Mp has more declared [parameters](#_Trm00059) than Mq, then Mp is better than Mq. This can occur if both [method](#_Trm00056)s have params [array](#_Trm00093)s and are applicable only in their [expanded form](#_Trm00238)s.
* Otherwise if all [parameters](#_Trm00059) of Mp have a corresponding argument whereas default [arguments](#_Trm00062) 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](#_Trm00011) than Mq, then Mp is better than Mq. Let {R1, R2, ..., Rn} and {S1, S2, ..., Sn} represent the uninstantiated and unexpanded parameter [types](#_Trm00011) of Mp and Mq. Mp's parameter [types](#_Trm00011) 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](#_Trm00178) is more specific than another [constructed type](#_Trm00178) (with the same number of type [arguments](#_Trm00062)) 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](#_Trm00093) type is more specific than another [array](#_Trm00093) type (with the same number of dimensions) if the [element type](#_Trm00095) of the first is more specific than the [element type](#_Trm00095) of the second.
* Otherwise if one member is a non-lifted [operator](#_Trm00090) and the other is a lifted [operator](#_Trm00090), the non-lifted one is better.
* Otherwise, neither function member is better.

#### Better [conversion](#_Trm00196) from expression

Given an [implicit](#_Trm00197) [conversion](#_Trm00196) C1 that converts from an expression E to a type T1, and an [implicit](#_Trm00197) [conversion](#_Trm00196) C2 that converts from an expression E to a type T2, C1 is a ***better conversion*** than C2 if at least one of the following holds:

* E has a type S and an identity [conversion](#_Trm00196) exists from S to T1 but not from S to T2
* E is not an anonymous function and T1 is a [better conversion](#_Trm00240) target than T2 ([§7.5.3.5](#_Toc00247))
* E is an anonymous function, T1 is either a [delegate type](#_Trm00107) D1 or an expression tree type Expression<D1>, T2 is either a [delegate type](#_Trm00107) D2 or an expression tree type Expression<D2> and one of the following holds:
  + D1 is a [better conversion](#_Trm00240) target than D2
  + D1 and D2 have identical parameter lists, and one of the following holds:
    - D1 has a [return type](#_Trm00060) Y1, and D2 has a [return type](#_Trm00060) Y2, an [inferred return type](#_Trm00233) X exists for E in the context of that parameter list ([§7.5.2.12](#_Toc00239)), and the [conversion](#_Trm00196) from X to Y1 is better than the [conversion](#_Trm00196) from X to Y2
    - E is async, D1 has a [return type](#_Trm00060) Task<Y1>, and D2 has a [return type](#_Trm00060) Task<Y2>, an [inferred return type](#_Trm00233) Task<X> exists for E in the context of that parameter list ([§7.5.2.12](#_Toc00239)), and the [conversion](#_Trm00196) from X to Y1 is better than the [conversion](#_Trm00196) from X to Y2
    - D1 has a [return type](#_Trm00060) Y, and D2 is void returning

#### Better [conversion](#_Trm00196) from type

Given a [conversion](#_Trm00196) C1 that converts from a type S to a type T1, and a [conversion](#_Trm00196) C2 that converts from a type S to a type T2, C1 is a ***better conversion*** than C2 if at least one of the following holds:

* An identity [conversion](#_Trm00196) exists from S to T1 but not from S to T2
* T1 is a [better conversion](#_Trm00240) target than T2 ([§7.5.3.5](#_Toc00247))

#### Better [conversion](#_Trm00196) target

Given two different [types](#_Trm00011) T1 and T2, T1 is a [better conversion](#_Trm00240) target than T2 if at least one of the following holds:

* An [implicit](#_Trm00197) [conversion](#_Trm00196) from T1 to T2 exists, and no [implicit](#_Trm00197) [conversion](#_Trm00196) from T2 to T1 exists
* T1 is a signed integral type and T2 is an unsigned integral type. Specifically:
  + T1 is sbyte and T2 is byte, ushort, uint, or ulong
  + T1 is short and T2 is ushort, uint, or ulong
  + T1 is int and T2 is uint, or ulong
  + T1 is long and T2 is ulong

#### Overloading in generic classes

While [signature](#_Trm00061)s as declared must be unique, it is possible that substitution of type [arguments](#_Trm00062) results in identical [signature](#_Trm00061)s. In such cases, the tie-breaking rules of [overload resolution](#_Trm00078) above will pick the [most specific](#_Trm00204) member.

The following examples show overloads that are valid and invalid according to this rule:

interface I1<T> {...}  
  
interface I2<T> {...}  
  
class G1<U>  
{  
 int F1(U u); // Overload resulotion for G<int>.F1  
 int F1(int i); // will pick non-generic  
  
 void F2(I1<U> a); // Valid overload  
 void F2(I2<U> a);  
}  
  
class G2<U,V>  
{  
 void F3(U u, V v); // Valid, but overload resolution for  
 void F3(V v, U u); // G2<int,int>.F3 will fail  
  
 void F4(U u, I1<V> v); // Valid, but overload resolution for  
 void F4(I1<V> v, U u); // G2<I1<int>,int>.F4 will fail  
  
 void F5(U u1, I1<V> v2); // Valid overload  
 void F5(V v1, U u2);  
  
 void F6(ref U u); // valid overload  
 void F6(out V v);  
}

### Compile-time checking of dynamic [overload resolution](#_Trm00078)

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](#_Trm00056) calls with dynamic [arguments](#_Trm00062)
* Instance [method](#_Trm00056) calls where the receiver is not a [dynamic expression](#_Trm00174)
* Indexer calls where the receiver is not a [dynamic expression](#_Trm00174)
* Constructor calls with dynamic [arguments](#_Trm00062)

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](#_Trm00231): Any type argument that does not depend directly or indirectly on an argument of type dynamic is inferred using the rules of [§7.5.2](#_Toc00227). The remaining type [arguments](#_Trm00062) are unknown.
* Partial applicability check: Applicability is checked according to [§7.5.3.1](#_Toc00243), but ignoring [parameters](#_Trm00059) whose [types](#_Trm00011) 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](#_Trm00210)-time process has already determined the particular member to invoke, possibly by applying [overload resolution](#_Trm00078) to a set of candidate [function members](#_Trm00079).

For purposes of describing the invocation process, [function members](#_Trm00079) are divided into two categories:

* Static [function members](#_Trm00079). These are [instance](#_Trm00172) constructors, static [method](#_Trm00056)s, static property [accessors](#_Trm00083), and user-[defined](#_Trm00121) [operator](#_Trm00090)s. Static [function members](#_Trm00079) are always non-virtual.
* Instance [function members](#_Trm00079). These are [instance](#_Trm00172) [method](#_Trm00056)s, [instance](#_Trm00172) property [accessors](#_Trm00083), and [indexer](#_Trm00087) [accessors](#_Trm00083). Instance [function members](#_Trm00079) are either non-virtual or virtual, and are always invoked on a particular [instance](#_Trm00172). The [instance](#_Trm00172) is computed by an [instance](#_Trm00172) expression, and it becomes [accessible](#_Trm00138) within the function member as this ([§7.6.7](#_Toc00267)).

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](#_Trm00172) member, E is the [instance](#_Trm00172) expression:

* If M is a static function member:
  + The argument list is evaluated as described in [§7.5.1](#_Toc00224).
  + M is invoked.
* If M is an [instance](#_Trm00172) function member declared in a [value\_type](#_Grm00029):
  + 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](#_Trm00193) of E's type is created and the [value](#_Trm00209) of E is assigned to that variable. E is then reclassified as a reference to that temporary [local variable](#_Trm00193). The temporary variable is [accessible](#_Trm00138) as this within M, but not in any other way. Thus, only when E is a true variable is it possible for the caller to observe the changes that M makes to this.
  + The argument list is evaluated as described in [§7.5.1](#_Toc00224).
  + M is invoked. The variable referenced by E becomes the variable referenced by this.
* If M is an [instance](#_Trm00172) function member declared in a [reference\_type](#_Grm00030):
  + E is evaluated. If this evaluation causes an exception, then no further steps are executed.
  + The argument list is evaluated as described in [§7.5.1](#_Toc00224).
  + If the type of E is a [value\_type](#_Grm00029), a [boxing](#_Trm00029) [conversion](#_Trm00196) ([§4.3.1](#_Toc00111)) 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](#_Trm00209) of E is checked to be valid. If the [value](#_Trm00209) 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](#_Trm00210)-time type of E is an [interface](#_Trm00102), the function member to invoke is the implementation of M provided by the [run-time type](#_Trm00073) of the [instance](#_Trm00172) referenced by E. This function member is determined by applying the [interface](#_Trm00102) mapping rules ([§13.4.4](#_Toc00547)) to determine the implementation of M provided by the [run-time type](#_Trm00073) of the [instance](#_Trm00172) 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](#_Trm00073) of the [instance](#_Trm00172) referenced by E. This function member is determined by applying the rules for determining the most derived implementation ([§10.6.3](#_Toc00448)) of M with respect to the [run-time type](#_Trm00073) of the [instance](#_Trm00172) 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](#_Trm00173) referenced by E becomes the [object](#_Trm00173) referenced by this.

#### Invocations on boxed [instance](#_Trm00172)s

A function member implemented in a [value\_type](#_Grm00029) can be invoked through a boxed [instance](#_Trm00172) of that [value\_type](#_Grm00029) in the following situations:

* When the function member is an override of a [method](#_Trm00056) [inherited](#_Trm00136) from type object and is invoked through an [instance](#_Trm00172) expression of type object.
* When the function member is an implementation of an [interface](#_Trm00102) function member and is invoked through an [instance](#_Trm00172) expression of an [interface\_type](#_Grm00030).
* When the function member is invoked through a delegate.

In these situations, the boxed [instance](#_Trm00172) is considered to contain a variable of the [value\_type](#_Grm00029), 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](#_Trm00172), it is possible for the function member to modify the [value](#_Trm00209) contained in the boxed [instance](#_Trm00172).

## Primary expressions

Primary expressions include the simplest forms of expressions.

primary\_expression:  
 | primary\_no\_array\_creation\_expression  
 | array\_creation\_expression  
 ;  
  
primary\_no\_array\_creation\_expression:  
 | literal  
 | simple\_name  
 | parenthesized\_expression  
 | member\_access  
 | invocation\_expression  
 | element\_access  
 | this\_access  
 | base\_access  
 | post\_increment\_expression  
 | post\_decrement\_expression  
 | object\_creation\_expression  
 | delegate\_creation\_expression  
 | anonymous\_object\_creation\_expression  
 | typeof\_expression  
 | checked\_expression  
 | unchecked\_expression  
 | default\_value\_expression  
 | anonymous\_method\_expression  
 | primary\_no\_array\_creation\_expression\_unsafe  
 ;

Primary expressions are divided between [array\_creation\_expression](#_Grm00047)s and [primary\_no\_array\_creation\_expression](#_Grm00035)s. Treating [array](#_Trm00093)-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](#_Grm00035) that consists of a [literal](#_Grm00009) ([§2.4.4](#_Toc00046)) is classified as a [value](#_Trm00209).

### Simple names

A [simple\_name](#_Grm00036) consists of an identifier, optionally followed by a type argument list:

simple\_name:  
 | identifier type\_argument\_list?  
 ;

A [simple\_name](#_Grm00036) 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](#_Grm00031). When no [type\_argument\_list](#_Grm00031) is specified, consider K to be zero. The [simple\_name](#_Grm00036) is evaluated and classified as follows:

* If K is zero and the [simple\_name](#_Grm00036) appears within a [block](#_Grm00071) and if the [block](#_Grm00071)'s (or an enclosing [block](#_Grm00071)'s) [local variable](#_Trm00193) [declaration space](#_Trm00130) ([§3.3](#_Toc00067)) contains a [local variable](#_Trm00193), parameter or constant with name I, then the [simple\_name](#_Grm00036) refers to that [local variable](#_Trm00193), parameter or constant and is classified as a variable or [value](#_Trm00209).
* If K is zero and the [simple\_name](#_Grm00036) appears within the body of a generic [method](#_Trm00056) declaration and if that declaration includes a type parameter with name I, then the [simple\_name](#_Grm00036) refers to that type parameter.
* Otherwise, for each [instance](#_Trm00172) type T ([§10.3.1](#_Toc00411)), starting with the [instance](#_Trm00172) type of the immediately enclosing type declaration and continuing with the [instance](#_Trm00172) 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](#_Grm00036) refers to that type parameter.
  + Otherwise, if a member lookup ([§7.4](#_Toc00221)) of I in T with K type [arguments](#_Trm00062) produces a match:
    - If T is the [instance](#_Trm00172) type of the immediately enclosing class or struct type and the lookup identifies one or more [method](#_Trm00056)s, the result is a [method](#_Trm00056) group with an associated [instance](#_Trm00172) expression of this. If a type argument list was specified, it is used in calling a generic [method](#_Trm00056) ([§7.6.5.1](#_Toc00261)).
    - Otherwise, if T is the [instance](#_Trm00172) type of the immediately enclosing class or struct type, if the lookup identifies an [instance](#_Trm00172) member, and if the reference occurs within the [block](#_Grm00071) of an [instance](#_Trm00172) constructor, an [instance](#_Trm00172) [method](#_Trm00056), or an [instance](#_Trm00172) accessor, the result is the same as a member access ([§7.6.4](#_Toc00257)) of the form this.I. This can only happen when K is zero.
    - Otherwise, the result is the same as a member access ([§7.6.4](#_Toc00257)) of the form T.I or T.I<A1,...,Ak>. In this case, it is a [binding](#_Trm00210)-time error for the [simple\_name](#_Grm00036) to refer to an [instance](#_Trm00172) member.
* Otherwise, for each namespace N, starting with the namespace in which the [simple\_name](#_Grm00036) occurs, continuing with each enclosing namespace (if any), and ending with the [global namespace](#_Trm00137), 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](#_Grm00036) occurs is enclosed by a namespace declaration for N and the namespace declaration contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) that associates the name I with a namespace or type, then the [simple\_name](#_Grm00036) is ambiguous and a compile-time error occurs.
    - Otherwise, the [simple\_name](#_Grm00036) refers to the namespace named I in N.
  + Otherwise, if N contains an [accessible](#_Trm00138) type having name I and K type [parameters](#_Trm00059), then:
    - If K is zero and the location where the [simple\_name](#_Grm00036) occurs is enclosed by a namespace declaration for N and the namespace declaration contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) that associates the name I with a namespace or type, then the [simple\_name](#_Grm00036) is ambiguous and a compile-time error occurs.
    - Otherwise, the [namespace\_or\_type\_name](#_Grm00027) refers to the type constructed with the given type [arguments](#_Trm00062).
  + Otherwise, if the location where the [simple\_name](#_Grm00036) occurs is enclosed by a namespace declaration for N:
    - If K is zero and the namespace declaration contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) that associates the name I with an imported namespace or type, then the [simple\_name](#_Grm00036) refers to that namespace or type.
    - Otherwise, if the [namespaces](#_Trm00010) imported by the [using\_namespace\_directive](#_Grm00103)s of the namespace declaration contain exactly one type having name I and K type [parameters](#_Trm00059), then the [simple\_name](#_Grm00036) refers to that type constructed with the given type [arguments](#_Trm00062).
    - Otherwise, if the [namespaces](#_Trm00010) imported by the [using\_namespace\_directive](#_Grm00103)s of the namespace declaration contain more than one type having name I and K type [parameters](#_Trm00059), then the [simple\_name](#_Grm00036) 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](#_Grm00027) ([§3.8](#_Toc00086)).

* Otherwise, the [simple\_name](#_Grm00036) is un[defined](#_Trm00121) and a compile-time error occurs.

#### Invariant meaning in [block](#_Trm00038)s

For each occurrence of a given identifier as a full [simple\_name](#_Grm00036) (without a type argument list) in an expression or declarator, within the [local variable](#_Trm00193) [declaration space](#_Trm00130) ([§3.3](#_Toc00067)) immediately enclosing that occurrence, every other occurrence of the same identifier as a full [simple\_name](#_Grm00036) in an expression or declarator must refer to the same entity. This rule ensures that the meaning of a name is always the same within a given [block](#_Trm00038), switch [block](#_Trm00038), for-, foreach- or using-statement, or anonymous function.

The example

class Test  
{  
 double x;  
  
 void F(bool b) {  
 x = 1.0;  
 if (b) {  
 int x;  
 x = 1;  
 }  
 }  
}

results in a compile-time error because x refers to different entities within the outer [block](#_Trm00038) (the extent of which includes the [nested](#_Trm00143) [block](#_Trm00038) in the if statement). In contrast, the example

class Test  
{  
 double x;  
  
 void F(bool b) {  
 if (b) {  
 x = 1.0;  
 }  
 else {  
 int x;  
 x = 1;  
 }  
 }  
}

is permitted because the name x is never used in the outer [block](#_Trm00038).

Note that the rule of invariant meaning applies only to simple names. It is perfectly valid for the same identifier to have one meaning as a simple name and another meaning as right operand of a member access ([§7.6.4](#_Toc00257)). For example:

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

The example above illustrates a common pattern of using the names of fields as parameter names in an [instance](#_Trm00172) constructor. In the example, the simple names x and y refer to the [parameters](#_Trm00059), but that does not pr[event](#_Trm00088) the member access expressions this.x and this.y from accessing the fields.

### Parenthesized expressions

A [parenthesized\_expression](#_Grm00037) consists of an [expression](#_Grm00067) enclosed in parentheses.

parenthesized\_expression:  
 | '(' expression ')'  
 ;

A [parenthesized\_expression](#_Grm00037) is evaluated by evaluating the [expression](#_Grm00067) within the parentheses. If the [expression](#_Grm00067) within the parentheses denotes a namespace or type, a compile-time error occurs. Otherwise, the result of the [parenthesized\_expression](#_Grm00037) is the result of the evaluation of the contained [expression](#_Grm00067).

### Member access

A [member\_access](#_Grm00038) consists of a [primary\_expression](#_Grm00035), a [predefined\_type](#_Grm00038), or a [qualified\_alias\_member](#_Grm00106), followed by a "." token, followed by an [identifier](#_Grm00007), optionally followed by a [type\_argument\_list](#_Grm00031).

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](#_Grm00106) production is [defined](#_Trm00121) in [§9.7](#_Toc00386).

A [member\_access](#_Grm00038) 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](#_Grm00031). When no [type\_argument\_list](#_Grm00031) is specified, consider K to be zero.

A [member\_access](#_Grm00038) with a [primary\_expression](#_Grm00035) of type dynamic is dynamically bound ([§7.2.2](#_Toc00209)). 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](#_Grm00038) are then applied at run-time, using the [run-time type](#_Trm00073) instead of the [compile-time type](#_Trm00074) of the [primary\_expression](#_Grm00035). If this run-time classification leads to a [method](#_Trm00056) group, then the member access must be the [primary\_expression](#_Grm00035) of an [invocation\_expression](#_Grm00039).

The [member\_access](#_Grm00038) is evaluated and classified as follows:

* If K is zero and E is a namespace and E contains a [nested](#_Trm00143) namespace with name I, then the result is that namespace.
* Otherwise, if E is a namespace and E contains an [accessible](#_Trm00138) type having name I and K type [parameters](#_Trm00059), then the result is that type constructed with the given type [arguments](#_Trm00062).
* If E is a [predefined\_type](#_Grm00038) or a [primary\_expression](#_Grm00035) classified as a type, if E is not a type parameter, and if a member lookup ([§7.4](#_Toc00221)) of I in E with K type [parameters](#_Trm00059) 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](#_Trm00062).
  + If I identifies one or more [method](#_Trm00056)s, then the result is a [method](#_Trm00056) group with no associated [instance](#_Trm00172) expression. If a type argument list was specified, it is used in calling a generic [method](#_Trm00056) ([§7.6.5.1](#_Toc00261)).
  + If I identifies a static property, then the result is a property access with no associated [instance](#_Trm00172) expression.
  + If I identifies a static field:
    - If the field is readonly and the reference occurs outside the [static constructor](#_Trm00081) of the class or struct in which the field is declared, then the result is a [value](#_Trm00209), namely the [value](#_Trm00209) of the [static field](#_Trm00053) I in E.
    - Otherwise, the result is a variable, namely the [static field](#_Trm00053) I in E.
  + If I identifies a static [event](#_Trm00088):
    - If the reference occurs within the class or struct in which the [event](#_Trm00088) is declared, and the [event](#_Trm00088) was declared without [event\_accessor\_declarations](#_Grm00120) ([§10.8](#_Toc00463)), then E.I is processed exactly as if I were a [static field](#_Trm00053).
    - Otherwise, the result is an [event](#_Trm00088) access with no associated [instance](#_Trm00172) expression.
  + If I identifies a constant, then the result is a [value](#_Trm00209), namely the [value](#_Trm00209) of that constant.
    - If I identifies an enumeration member, then the result is a [value](#_Trm00209), namely the [value](#_Trm00209) 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](#_Trm00087) access, variable, or [value](#_Trm00209), the type of which is T, and a member lookup ([§7.4](#_Toc00221)) of I in T with K type [arguments](#_Trm00062) produces a match, then E.I is evaluated and classified as follows:
  + First, if E is a property or [indexer](#_Trm00087) access, then the [value](#_Trm00209) of the property or [indexer](#_Trm00087) access is obtained ([§7.1.1](#_Toc00206)) and E is reclassified as a [value](#_Trm00209).
  + If I identifies one or more [method](#_Trm00056)s, then the result is a [method](#_Trm00056) group with an associated [instance](#_Trm00172) expression of E. If a type argument list was specified, it is used in calling a generic [method](#_Trm00056) ([§7.6.5.1](#_Toc00261)).
  + If I identifies an [instance](#_Trm00172) property, then the result is a property access with an associated [instance](#_Trm00172) expression of E.
  + If T is a [class\_type](#_Grm00030) and I identifies an [instance](#_Trm00172) field of that [class\_type](#_Grm00030):
    - If the [value](#_Trm00209) of E is null, then a System.NullReferenceException is thrown.
    - Otherwise, if the field is readonly and the reference occurs outside an [instance](#_Trm00172) constructor of the class in which the field is declared, then the result is a [value](#_Trm00209), namely the [value](#_Trm00209) of the field I in the [object](#_Trm00173) referenced by E.
    - Otherwise, the result is a variable, namely the field I in the [object](#_Trm00173) referenced by E.
  + If T is a [struct\_type](#_Grm00029) and I identifies an [instance](#_Trm00172) field of that [struct\_type](#_Grm00029):
    - If E is a [value](#_Trm00209), or if the field is readonly and the reference occurs outside an [instance](#_Trm00172) constructor of the struct in which the field is declared, then the result is a [value](#_Trm00209), namely the [value](#_Trm00209) of the field I in the struct [instance](#_Trm00172) given by E.
    - Otherwise, the result is a variable, namely the field I in the struct [instance](#_Trm00172) given by E.
  + If I identifies an [instance](#_Trm00172) [event](#_Trm00088):
    - If the reference occurs within the class or struct in which the [event](#_Trm00088) is declared, and the [event](#_Trm00088) was declared without [event\_accessor\_declarations](#_Grm00120) ([§10.8](#_Toc00463)), and the reference does not occur as the left-hand side of a += or -= [operator](#_Trm00090), then E.I is processed exactly as if I was an [instance](#_Trm00172) field.
    - Otherwise, the result is an [event](#_Trm00088) access with an associated [instance](#_Trm00172) expression of E.
* Otherwise, an attempt is made to process E.I as an extension [method](#_Trm00056) invocation ([§7.6.5.2](#_Toc00262)). If this fails, E.I is an invalid member reference, and a [binding](#_Trm00210)-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](#_Grm00036) ([§7.6.2](#_Toc00254)) is a constant, field, property, [local variable](#_Trm00193), or parameter with the same type as the meaning of E as a [type\_name](#_Grm00027) ([§3.8](#_Toc00086)), 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](#_Trm00012) and [nested](#_Trm00143) [types](#_Trm00011) 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](#_Grm00036) ([§7.6.2](#_Toc00254)) and [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) 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](#_Trm00062), 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](#_Trm00056) G with two type [arguments](#_Trm00062) and one regular argument.

If a sequence of tokens can be parsed (in context) as a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)), [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)), or [pointer\_member\_access](#_Grm00153) ([§18.5.2](#_Toc00598)) ending with a [type\_argument\_list](#_Grm00031) ([§4.4.1](#_Toc00114)), the token immediately following the closing > token is examined. If it is one of

( ) ] } : ; , . ? == != | ^

then the [type\_argument\_list](#_Grm00031) is retained as part of the [simple\_name](#_Grm00036), [member\_access](#_Grm00038) or [pointer\_member\_access](#_Grm00153) and any other possible parse of the sequence of tokens is discarded. Otherwise, the [type\_argument\_list](#_Grm00031) is not considered to be part of the [simple\_name](#_Grm00036), [member\_access](#_Grm00038) or [pointer\_member\_access](#_Grm00153), 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](#_Grm00031) in a [namespace\_or\_type\_name](#_Grm00027) ([§3.8](#_Toc00086)). 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](#_Trm00056) G with two type [arguments](#_Trm00062) and one regular argument. The [statements](#_Trm00037)

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

will each be interpreted as a call to F with two [arguments](#_Trm00062). The statement

x = F < A > +y;

will be interpreted as a less than [operator](#_Trm00090), greater than [operator](#_Trm00090), and unary plus [operator](#_Trm00090), as if the statement had been written x = (F < A) > (+y), instead of as a [simple\_name](#_Grm00036) with a [type\_argument\_list](#_Grm00031) followed by a binary plus [operator](#_Trm00090). In the statement

x = y is C<T> + z;

the tokens C<T> are interpreted as a [namespace\_or\_type\_name](#_Grm00027) with a [type\_argument\_list](#_Grm00031).

### Invocation expressions

An [invocation\_expression](#_Grm00039) is used to invoke a [method](#_Trm00056).

invocation\_expression:  
 | primary\_expression '(' argument\_list? ')'  
 ;

An [invocation\_expression](#_Grm00039) is dynamically bound ([§7.2.2](#_Toc00209)) if at least one of the following holds:

* The [primary\_expression](#_Grm00035) has [compile-time type](#_Trm00074) dynamic.
* At least one argument of the optional [argument\_list](#_Grm00034) has [compile-time type](#_Trm00074) dynamic and the [primary\_expression](#_Grm00035) does not have a [delegate type](#_Trm00107).

In this case the compiler classifies the [invocation\_expression](#_Grm00039) as a [value](#_Trm00209) of type dynamic. The rules below to determine the meaning of the [invocation\_expression](#_Grm00039) are then applied at run-time, using the [run-time type](#_Trm00073) instead of the [compile-time type](#_Trm00074) of those of the [primary\_expression](#_Grm00035) and [arguments](#_Trm00062) which have the [compile-time type](#_Trm00074) dynamic. If the [primary\_expression](#_Grm00035) does not have [compile-time type](#_Trm00074) dynamic, then the [method](#_Trm00056) invocation undergoes a limited compile time check as described in [§7.5.4](#_Toc00249).

The [primary\_expression](#_Grm00035) of an [invocation\_expression](#_Grm00039) must be a [method](#_Trm00056) group or a [value](#_Trm00209) of a [delegate\_type](#_Grm00030). If the [primary\_expression](#_Grm00035) is a [method](#_Trm00056) group, the [invocation\_expression](#_Grm00039) is a [method](#_Trm00056) invocation ([§7.6.5.1](#_Toc00261)). If the [primary\_expression](#_Grm00035) is a [value](#_Trm00209) of a [delegate\_type](#_Grm00030), the [invocation\_expression](#_Grm00039) is a delegate invocation ([§7.6.5.3](#_Toc00263)). If the [primary\_expression](#_Grm00035) is neither a [method](#_Trm00056) group nor a [value](#_Trm00209) of a [delegate\_type](#_Grm00030), a [binding](#_Trm00210)-time error occurs.

The optional [argument\_list](#_Grm00034) ([§7.5.1](#_Toc00224)) provides [value](#_Trm00209)s or variable [references](#_Trm00160) for the [parameters](#_Trm00059) of the [method](#_Trm00056).

The result of evaluating an [invocation\_expression](#_Grm00039) is classified as follows:

* If the [invocation\_expression](#_Grm00039) invokes a [method](#_Trm00056) 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](#_Grm00078) ([§8.6](#_Toc00357)) or as the body of a [lambda\_expression](#_Grm00064) ([§7.15](#_Toc00321)). Otherwise a [binding](#_Trm00210)-time error occurs.
* Otherwise, the result is a [value](#_Trm00209) of the type returned by the [method](#_Trm00056) or delegate.

#### Method invocations

For a [method](#_Trm00056) invocation, the [primary\_expression](#_Grm00035) of the [invocation\_expression](#_Grm00039) must be a [method](#_Trm00056) group. The [method](#_Trm00056) group identifies the one [method](#_Trm00056) to invoke or the set of [overloaded](#_Trm00036) [method](#_Trm00056)s from which to choose a specific [method](#_Trm00056) to invoke. In the latter case, determination of the specific [method](#_Trm00056) to invoke is based on the context provided by the [types](#_Trm00011) of the [arguments](#_Trm00062) in the [argument\_list](#_Grm00034).

The [binding](#_Trm00210)-time processing of a [method](#_Trm00056) invocation of the form M(A), where M is a [method](#_Trm00056) group (possibly including a [type\_argument\_list](#_Grm00031)), and A is an optional [argument\_list](#_Grm00034), consists of the following steps:

* The set of candidate [method](#_Trm00056)s for the [method](#_Trm00056) invocation is constructed. For each [method](#_Trm00056) F associated with the [method](#_Trm00056) group M:
  + If F is non-generic, F is a candidate when:
    - M has no type argument list, and
    - F is applicable with respect to A ([§7.5.3.1](#_Toc00243)).
  + If F is generic and M has no type argument list, F is a candidate when:
    - Type inference ([§7.5.2](#_Toc00227)) succeeds, inferring a list of type [arguments](#_Trm00062) for the call, and
    - Once the inferred type [arguments](#_Trm00062) are substituted for the corresponding [method](#_Trm00056) type [parameters](#_Trm00059), all [constructed type](#_Trm00178)s in the parameter list of F satisfy their constraints ([§4.4.4](#_Toc00117)), and the parameter list of F is applicable with respect to A ([§7.5.3.1](#_Toc00243)).
  + If F is generic and M includes a type argument list, F is a candidate when:
    - F has the same number of [method](#_Trm00056) type [parameters](#_Trm00059) as were supplied in the type argument list, and
    - Once the type [arguments](#_Trm00062) are substituted for the corresponding [method](#_Trm00056) type [parameters](#_Trm00059), all [constructed type](#_Trm00178)s in the parameter list of F satisfy their constraints ([§4.4.4](#_Toc00117)), and the parameter list of F is applicable with respect to A ([§7.5.3.1](#_Toc00243)).
* The set of candidate [method](#_Trm00056)s is reduced to contain only [method](#_Trm00056)s from the most derived [types](#_Trm00011): For each [method](#_Trm00056) C.F in the set, where C is the type in which the [method](#_Trm00056) F is declared, all [method](#_Trm00056)s declared in a base type of C are removed from the set. Furthermore, if C is a class type other than object, all [method](#_Trm00056)s declared in an [interface](#_Trm00102) type are removed from the set. (This latter rule only has affect when the [method](#_Trm00056) group was the result of a member lookup on a type parameter having an effective base class other than [object](#_Trm00173) and a non-empty effective [interface](#_Trm00102) set.)
* If the resulting set of candidate [method](#_Trm00056)s 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](#_Trm00056) invocation ([§7.6.5.2](#_Toc00262)). If this fails, then no applicable [method](#_Trm00056)s exist, and a [binding](#_Trm00210)-time error occurs.
* The best [method](#_Trm00056) of the set of candidate [method](#_Trm00056)s is identified using the [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242). If a single best [method](#_Trm00056) cannot be identified, the [method](#_Trm00056) invocation is ambiguous, and a [binding](#_Trm00210)-time error occurs. When performing [overload resolution](#_Trm00078), the [parameters](#_Trm00059) of a generic [method](#_Trm00056) are considered after substituting the type [arguments](#_Trm00062) (supplied or inferred) for the corresponding [method](#_Trm00056) type [parameters](#_Trm00059).
* Final validation of the chosen best [method](#_Trm00056) is performed:
  + The [method](#_Trm00056) is validated in the context of the [method](#_Trm00056) group: If the best [method](#_Trm00056) is a static [method](#_Trm00056), the [method](#_Trm00056) group must have resulted from a [simple\_name](#_Grm00036) or a [member\_access](#_Grm00038) through a type. If the best [method](#_Trm00056) is an [instance](#_Trm00172) [method](#_Trm00056), the [method](#_Trm00056) group must have resulted from a [simple\_name](#_Grm00036), a [member\_access](#_Grm00038) through a variable or [value](#_Trm00209), or a [base\_access](#_Grm00042). If neither of these requirements is true, a [binding](#_Trm00210)-time error occurs.
  + If the best [method](#_Trm00056) is a generic [method](#_Trm00056), the type [arguments](#_Trm00062) (supplied or inferred) are checked against the constraints ([§4.4.4](#_Toc00117)) declared on the generic [method](#_Trm00056). If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a [binding](#_Trm00210)-time error occurs.

Once a [method](#_Trm00056) has been selected and validated at [binding](#_Trm00210)-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in [§7.5.4](#_Toc00249).

The intuitive effect of the resolution rules described above is as follows: To locate the particular [method](#_Trm00056) invoked by a [method](#_Trm00056) invocation, start with the type indicated by the [method](#_Trm00056) invocation and proceed up the [inheritance](#_Trm00047) chain until at least one applicable, [accessible](#_Trm00138), non-override [method](#_Trm00056) declaration is found. Then perform [type inference](#_Trm00231) and [overload resolution](#_Trm00078) on the set of applicable, [accessible](#_Trm00138), non-override [method](#_Trm00056)s declared in that type and invoke the [method](#_Trm00056) thus selected. If no [method](#_Trm00056) was found, try instead to process the invocation as an extension [method](#_Trm00056) invocation.

#### Extension [method](#_Trm00056) invocations

In a [method](#_Trm00056) invocation ([§7.5.5.1](#_Toc00251)) 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 [method](#_Trm00056)s, an attempt is made to process the construct as an extension [method](#_Trm00056) invocation. If *expr* or any of the *args* has [compile-time type](#_Trm00074) dynamic, extension [method](#_Trm00056)s will not apply.

The [object](#_Trm00173)ive is to find the best [type\_name](#_Grm00027) C, so that the corresponding static [method](#_Trm00056) invocation can take place:

C . identifier ( expr )  
  
C . identifier ( expr , args )  
  
C . identifier < typeargs > ( expr )  
  
C . identifier < typeargs > ( expr , args )

An extension [method](#_Trm00056) Ci.Mj is ***eligible*** if:

* Ci is a non-generic, non-[nested](#_Trm00143) class
* The name of Mj is [identifier](#_Grm00007)
* Mj is [accessible](#_Trm00138) and applicable when applied to the [arguments](#_Trm00062) as a static [method](#_Trm00056) as shown above
* An [implicit](#_Trm00197) identity, reference or [boxing](#_Trm00029) [conversion](#_Trm00196) 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 [method](#_Trm00056)s:
  + If the given namespace or compilation unit directly contains non-generic [type declarations](#_Trm00028) Ci with [eligible](#_Trm00242) extension [method](#_Trm00056)s Mj, then the set of those extension [method](#_Trm00056)s is the candidate set.
  + If [namespaces](#_Trm00010) imported by using namespace directives in the given namespace or compilation unit directly contain non-generic [type declarations](#_Trm00028) Ci with [eligible](#_Trm00242) extension [method](#_Trm00056)s Mj, then the set of those extension [method](#_Trm00056)s 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](#_Trm00078) is applied to the candidate set as described in ([§7.5.3](#_Toc00242)). If no single best [method](#_Trm00056) is found, a compile-time error occurs.
* C is the type within which the best [method](#_Trm00056) is declared as an extension [method](#_Trm00056).

Using C as a target, the [method](#_Trm00056) call is then processed as a static [method](#_Trm00056) invocation ([§7.5.4](#_Toc00249)).

The preceding rules mean that [instance](#_Trm00172) [method](#_Trm00056)s take [precedence](#_Trm00035) over extension [method](#_Trm00056)s, that extension [method](#_Trm00056)s available in inner namespace declarations take [precedence](#_Trm00035) over extension [method](#_Trm00056)s available in outer namespace declarations, and that extension [method](#_Trm00056)s declared directly in a namespace take [precedence](#_Trm00035) over extension [method](#_Trm00056)s 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](#_Trm00056) takes [precedence](#_Trm00035) over the first extension [method](#_Trm00056), and C's [method](#_Trm00056) takes [precedence](#_Trm00035) over both extension [method](#_Trm00056)s.

public static class C  
{  
 public static void F(this int i) { Console.WriteLine("C.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("C.G({0})", i); }  
 public static void H(this int i) { Console.WriteLine("C.H({0})", i); }  
}  
  
namespace N1  
{  
 public static class D  
 {  
 public static void F(this int i) { Console.WriteLine("D.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("D.G({0})", i); }  
 }  
}  
  
namespace N2  
{  
 using N1;  
  
 public static class E  
 {  
 public static void F(this int i) { Console.WriteLine("E.F({0})", i); }  
 }  
  
 class Test  
 {  
 static void Main(string[] args)  
 {  
 1.F();  
 2.G();  
 3.H();  
 }  
 }  
}

The output of this example is:

E.F(1)  
D.G(2)  
C.H(3)

D.G takes precendece over C.G, and E.F takes [precedence](#_Trm00035) over both D.F and C.F.

#### Delegate invocations

For a delegate invocation, the [primary\_expression](#_Grm00035) of the [invocation\_expression](#_Grm00039) must be a [value](#_Trm00209) of a [delegate\_type](#_Grm00030). Furthermore, considering the [delegate\_type](#_Grm00030) to be a function member with the same parameter list as the [delegate\_type](#_Grm00030), the [delegate\_type](#_Grm00030) must be applicable ([§7.5.3.1](#_Toc00243)) with respect to the [argument\_list](#_Grm00034) of the [invocation\_expression](#_Grm00039).

The run-time processing of a delegate invocation of the form D(A), where D is a [primary\_expression](#_Grm00035) of a [delegate\_type](#_Grm00030) and A is an optional [argument\_list](#_Grm00034), consists of the following steps:

* D is evaluated. If this evaluation causes an exception, no further steps are executed.
* The [value](#_Trm00209) of D is checked to be valid. If the [value](#_Trm00209) of D is null, a System.NullReferenceException is thrown and no further steps are executed.
* Otherwise, D is a reference to a delegate [instance](#_Trm00172). Function member invocations ([§7.5.4](#_Toc00249)) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an [instance](#_Trm00172) and [instance](#_Trm00172) [method](#_Trm00056), the [instance](#_Trm00172) for the invocation is the [instance](#_Trm00172) contained in the callable entity.

### Element access

An [element\_access](#_Grm00040) consists of a [primary\_no\_array\_creation\_expression](#_Grm00035), followed by a "[" token, followed by an [argument\_list](#_Grm00034), followed by a "]" token. The [argument\_list](#_Grm00034) consists of one or more [argument](#_Grm00034)s, separated by commas.

element\_access:  
 | primary\_no\_array\_creation\_expression '[' expression\_list ']'  
 ;

The [argument\_list](#_Grm00034) of an [element\_access](#_Grm00040) is not allowed to contain ref or out [arguments](#_Trm00062).

An [element\_access](#_Grm00040) is dynamically bound ([§7.2.2](#_Toc00209)) if at least one of the following holds:

* The [primary\_no\_array\_creation\_expression](#_Grm00035) has [compile-time type](#_Trm00074) dynamic.
* At least one expression of the [argument\_list](#_Grm00034) has [compile-time type](#_Trm00074) dynamic and the [primary\_no\_array\_creation\_expression](#_Grm00035) does not have an [array](#_Trm00093) type.

In this case the compiler classifies the [element\_access](#_Grm00040) as a [value](#_Trm00209) of type dynamic. The rules below to determine the meaning of the [element\_access](#_Grm00040) are then applied at run-time, using the [run-time type](#_Trm00073) instead of the [compile-time type](#_Trm00074) of those of the [primary\_no\_array\_creation\_expression](#_Grm00035) and [argument\_list](#_Grm00034) expressions which have the [compile-time type](#_Trm00074) dynamic. If the [primary\_no\_array\_creation\_expression](#_Grm00035) does not have [compile-time type](#_Trm00074) dynamic, then the element access undergoes a limited compile time check as described in [§7.5.4](#_Toc00249).

If the [primary\_no\_array\_creation\_expression](#_Grm00035) of an [element\_access](#_Grm00040) is a [value](#_Trm00209) of an [array\_type](#_Grm00030), the [element\_access](#_Grm00040) is an [array](#_Trm00093) access ([§7.6.6.1](#_Toc00265)). Otherwise, the [primary\_no\_array\_creation\_expression](#_Grm00035) must be a variable or [value](#_Trm00209) of a class, struct, or [interface](#_Trm00102) type that has one or more [indexer](#_Trm00087) [members](#_Trm00012), in which case the [element\_access](#_Grm00040) is an [indexer](#_Trm00087) access ([§7.6.6.2](#_Toc00266)).

#### Array access

For an [array](#_Trm00093) access, the [primary\_no\_array\_creation\_expression](#_Grm00035) of the [element\_access](#_Grm00040) must be a [value](#_Trm00209) of an [array\_type](#_Grm00030). Furthermore, the [argument\_list](#_Grm00034) of an [array](#_Trm00093) access is not allowed to contain named [arguments](#_Trm00062).The number of expressions in the [argument\_list](#_Grm00034) must be the same as the [rank](#_Trm00099) of the [array\_type](#_Grm00030), and each expression must be of type int, uint, long, ulong, or must be [implicit](#_Trm00197)ly convertible to one or more of these [types](#_Trm00011).

The result of evaluating an [array](#_Trm00093) access is a variable of the [element type](#_Trm00095) of the [array](#_Trm00093), namely the [array](#_Trm00093) element selected by the [value](#_Trm00209)(s) of the expression(s) in the [argument\_list](#_Grm00034).

The run-time processing of an [array](#_Trm00093) access of the form P[A], where P is a [primary\_no\_array\_creation\_expression](#_Grm00035) of an [array\_type](#_Grm00030) and A is an [argument\_list](#_Grm00034), 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](#_Grm00034) are evaluated in order, from left to right. Following evaluation of each index expression, an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) to one of the following [types](#_Trm00011) is performed: int, uint, long, ulong. The first type in this list for which an [implicit](#_Trm00197) [conversion](#_Trm00196) exists is chosen. For [instance](#_Trm00172), if the index expression is of type short then an [implicit](#_Trm00197) [conversion](#_Trm00196) to int is performed, since [implicit](#_Trm00197) [conversion](#_Trm00196)s from short to int and from short to long are possible. If evaluation of an index expression or the subsequent [implicit](#_Trm00197) [conversion](#_Trm00196) causes an exception, then no further index expressions are evaluated and no further steps are executed.
* The [value](#_Trm00209) of P is checked to be valid. If the [value](#_Trm00209) of P is null, a System.NullReferenceException is thrown and no further steps are executed.
* The [value](#_Trm00209) of each expression in the [argument\_list](#_Grm00034) is checked against the actual bounds of each dimension of the [array](#_Trm00093) [instance](#_Trm00172) referenced by P. If one or more [value](#_Trm00209)s are out of range, a System.IndexOutOfRangeException is thrown and no further steps are executed.
* The location of the [array](#_Trm00093) element given by the index expression(s) is computed, and this location becomes the result of the [array](#_Trm00093) access.

#### Indexer access

For an [indexer](#_Trm00087) access, the [primary\_no\_array\_creation\_expression](#_Grm00035) of the [element\_access](#_Grm00040) must be a variable or [value](#_Trm00209) of a class, struct, or [interface](#_Trm00102) type, and this type must implement one or more [indexer](#_Trm00087)s that are applicable with respect to the [argument\_list](#_Grm00034) of the [element\_access](#_Grm00040).

The [binding](#_Trm00210)-time processing of an [indexer](#_Trm00087) access of the form P[A], where P is a [primary\_no\_array\_creation\_expression](#_Grm00035) of a class, struct, or [interface](#_Trm00102) type T, and A is an [argument\_list](#_Grm00034), consists of the following steps:

* The set of [indexer](#_Trm00087)s provided by T is constructed. The set consists of all [indexer](#_Trm00087)s declared in T or a base type of T that are not override declarations and are [accessible](#_Trm00138) in the current context ([§3.5](#_Toc00076)).
* The set is reduced to those [indexer](#_Trm00087)s that are applicable and not [hidden](#_Trm00150) by other [indexer](#_Trm00087)s. The following rules are applied to each [indexer](#_Trm00087) S.I in the set, where S is the type in which the [indexer](#_Trm00087) I is declared:
  + If I is not applicable with respect to A ([§7.5.3.1](#_Toc00243)), then I is removed from the set.
  + If I is applicable with respect to A ([§7.5.3.1](#_Toc00243)), then all [indexer](#_Trm00087)s declared in a base type of S are removed from the set.
  + If I is applicable with respect to A ([§7.5.3.1](#_Toc00243)) and S is a class type other than object, all [indexer](#_Trm00087)s declared in an [interface](#_Trm00102) are removed from the set.
* If the resulting set of candidate [indexer](#_Trm00087)s is empty, then no applicable [indexer](#_Trm00087)s exist, and a [binding](#_Trm00210)-time error occurs.
* The best [indexer](#_Trm00087) of the set of candidate [indexer](#_Trm00087)s is identified using the [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242). If a single best [indexer](#_Trm00087) cannot be identified, the [indexer](#_Trm00087) access is ambiguous, and a [binding](#_Trm00210)-time error occurs.
* The index expressions of the [argument\_list](#_Grm00034) are evaluated in order, from left to right. The result of processing the [indexer](#_Trm00087) access is an expression classified as an [indexer](#_Trm00087) access. The [indexer](#_Trm00087) access expression [references](#_Trm00160) the [indexer](#_Trm00087) determined in the step above, and has an associated [instance](#_Trm00172) expression of P and an associated argument list of A.

Depending on the context in which it is used, an [indexer](#_Trm00087) access causes invocation of either the *get accessor* or the *set accessor* of the [indexer](#_Trm00087). If the [indexer](#_Trm00087) access is the target of an assignment, the *set accessor* is invoked to assign a new [value](#_Trm00209) ([§7.17.1](#_Toc00342)). In all other cases, the *get accessor* is invoked to obtain the current [value](#_Trm00209) ([§7.1.1](#_Toc00206)).

### This access

A [this\_access](#_Grm00041) consists of the reserved word this.

this\_access:  
 | 'this'  
 ;

A [this\_access](#_Grm00041) is permitted only in the [block](#_Grm00071) of an [instance](#_Trm00172) constructor, an [instance](#_Trm00172) [method](#_Trm00056), or an [instance](#_Trm00172) accessor. It has one of the following meanings:

* When this is used in a [primary\_expression](#_Grm00035) within an [instance](#_Trm00172) constructor of a class, it is classified as a [value](#_Trm00209). The type of the [value](#_Trm00209) is the [instance](#_Trm00172) type ([§10.3.1](#_Toc00411)) of the class within which the usage occurs, and the [value](#_Trm00209) is a reference to the [object](#_Trm00173) being constructed.
* When this is used in a [primary\_expression](#_Grm00035) within an [instance](#_Trm00172) [method](#_Trm00056) or [instance](#_Trm00172) accessor of a class, it is classified as a [value](#_Trm00209). The type of the [value](#_Trm00209) is the [instance](#_Trm00172) type ([§10.3.1](#_Toc00411)) of the class within which the usage occurs, and the [value](#_Trm00209) is a reference to the [object](#_Trm00173) for which the [method](#_Trm00056) or accessor was invoked.
* When this is used in a [primary\_expression](#_Grm00035) within an [instance](#_Trm00172) constructor of a struct, it is classified as a variable. The type of the variable is the [instance](#_Trm00172) type ([§10.3.1](#_Toc00411)) of the struct within which the usage occurs, and the variable represents the struct being constructed. The this variable of an [instance](#_Trm00172) 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](#_Trm00068) in every execution path of the [instance](#_Trm00172) constructor.
* When this is used in a [primary\_expression](#_Grm00035) within an [instance](#_Trm00172) [method](#_Trm00056) or [instance](#_Trm00172) accessor of a struct, it is classified as a variable. The type of the variable is the [instance](#_Trm00172) type ([§10.3.1](#_Toc00411)) of the struct within which the usage occurs.
  + If the [method](#_Trm00056) or accessor is not an iterator ([§10.14](#_Toc00483)), the this variable represents the struct for which the [method](#_Trm00056) or accessor was invoked, and behaves exactly the same as a ref parameter of the struct type.
  + If the [method](#_Trm00056) or accessor is an iterator, the this variable represents a copy of the struct for which the [method](#_Trm00056) or accessor was invoked, and behaves exactly the same as a [value](#_Trm00209) parameter of the struct type.

Use of this in a [primary\_expression](#_Grm00035) 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](#_Trm00056), a static property accessor, or in a [variable\_initializer](#_Grm00115) of a field declaration.

### Base access

A [base\_access](#_Grm00042) consists of the reserved word base followed by either a "." token and an identifier or an [argument\_list](#_Grm00034) enclosed in square brackets:

base\_access:  
 | 'base' '.' identifier  
 | 'base' '[' expression\_list ']'  
 ;

A [base\_access](#_Grm00042) is used to access base class [members](#_Trm00012) that are [hidden](#_Trm00150) by similarly named [members](#_Trm00012) in the current class or struct. A [base\_access](#_Grm00042) is permitted only in the [block](#_Grm00071) of an [instance](#_Trm00172) constructor, an [instance](#_Trm00172) [method](#_Trm00056), or an [instance](#_Trm00172) 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](#_Trm00087) must exist in the base class.

At [binding](#_Trm00210)-time, [base\_access](#_Grm00042) 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](#_Trm00172) of the base class.

When a [base\_access](#_Grm00042) [references](#_Trm00160) a virtual function member (a [method](#_Trm00056), property, or [indexer](#_Trm00087)), the determination of which function member to invoke at run-time ([§7.5.4](#_Toc00249)) is changed. The function member that is invoked is determined by finding the most derived implementation ([§10.6.3](#_Toc00448)) of the function member with respect to B (instead of with respect to the [run-time type](#_Trm00073) of this, as would be usual in a non-base access). Thus, within an override of a virtual function member, a [base\_access](#_Grm00042) can be used to invoke the [inherited](#_Trm00136) implementation of the function member. If the function member referenced by a [base\_access](#_Grm00042) is [abstract](#_Trm00076), a [binding](#_Trm00210)-time error occurs.

### Postfix increment and decrement [operator](#_Trm00090)s

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](#_Trm00087) access. The result of the operation is a [value](#_Trm00209) of the same type as the operand.

If the [primary\_expression](#_Grm00035) has the [compile-time type](#_Trm00074) dynamic then the [operator](#_Trm00090) is dynamically bound ([§7.2.2](#_Toc00209)), the [post\_increment\_expression](#_Grm00043) or [post\_decrement\_expression](#_Grm00043) has the [compile-time type](#_Trm00074) dynamic and the following rules are applied at run-time using the [run-time type](#_Trm00073) of the [primary\_expression](#_Grm00035).

If the operand of a postfix increment or decrement operation is a property or [indexer](#_Trm00087) access, the property or [indexer](#_Trm00087) must have both a get and a set accessor. If this is not the case, a [binding](#_Trm00210)-time error occurs.

Unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is applied to select a specific [operator](#_Trm00090) implementation. Pre[defined](#_Trm00121) ++ and -- [operator](#_Trm00090)s exist for the following [types](#_Trm00011): sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any [enum type](#_Trm00105). The pre[defined](#_Trm00121) ++ [operator](#_Trm00090)s return the [value](#_Trm00209) produced by adding 1 to the operand, and the pre[defined](#_Trm00121) -- [operator](#_Trm00090)s return the [value](#_Trm00209) 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](#_Trm00105), 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](#_Trm00209) of x is saved.
  + The selected [operator](#_Trm00090) is invoked with the saved [value](#_Trm00209) of x as its argument.
  + The [value](#_Trm00209) returned by the [operator](#_Trm00090) is stored in the location given by the evaluation of x.
  + The saved [value](#_Trm00209) of x becomes the result of the operation.
* If x is classified as a property or [indexer](#_Trm00087) access:
  + The [instance](#_Trm00172) expression (if x is not static) and the argument list (if x is an [indexer](#_Trm00087) 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](#_Trm00209) is saved.
  + The selected [operator](#_Trm00090) is invoked with the saved [value](#_Trm00209) of x as its argument.
  + The set accessor of x is invoked with the [value](#_Trm00209) returned by the [operator](#_Trm00090) as its value argument.
  + The saved [value](#_Trm00209) of x becomes the result of the operation.

The ++ and -- [operator](#_Trm00090)s also support prefix notation ([§7.7.5](#_Toc00286)). Typically, the result of x++ or x-- is the [value](#_Trm00209) of x before the operation, whereas the result of ++x or --x is the [value](#_Trm00209) of x after the operation. In either case, x itself has the same [value](#_Trm00209) 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](#_Trm00090) implementations for the two notations.

### The new [operator](#_Trm00090)

The new [operator](#_Trm00090) is used to create new [instance](#_Trm00172)s of [types](#_Trm00011).

There are three forms of new expressions:

* Object creation expressions are used to create new [instance](#_Trm00172)s of [class types](#_Trm00024) and [value](#_Trm00209) [types](#_Trm00011).
* Array creation expressions are used to create new [instance](#_Trm00172)s of [array](#_Trm00093) [types](#_Trm00011).
* Delegate creation expressions are used to create new [instance](#_Trm00172)s of [delegate type](#_Trm00107)s.

The new [operator](#_Trm00090) implies creation of an [instance](#_Trm00172) of a type, but does not necessarily imply dynamic allocation of memory. In particular, [instance](#_Trm00172)s of [value](#_Trm00209) [types](#_Trm00011) require no additional memory beyond the [variables](#_Trm00031) in which they reside, and no dynamic allocations occur when new is used to create [instance](#_Trm00172)s of [value](#_Trm00209) [types](#_Trm00011).

#### Object creation expressions

An [object\_creation\_expression](#_Grm00044) is used to create a new [instance](#_Trm00172) of a [class\_type](#_Grm00030) or a [value\_type](#_Grm00029).

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](#_Grm00028) of an [object\_creation\_expression](#_Grm00044) must be a [class\_type](#_Grm00030), a [value\_type](#_Grm00029) or a [type\_parameter](#_Grm00032). The [type](#_Grm00028) cannot be an abstract [class\_type](#_Grm00030).

The optional [argument\_list](#_Grm00034) ([§7.5.1](#_Toc00224)) is permitted only if the [type](#_Grm00028) is a [class\_type](#_Grm00030) or a [struct\_type](#_Grm00029).

An [object](#_Trm00173) creation expression can omit the constructor argument list and enclosing parentheses provided it includes an [object](#_Trm00173) initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Processing of an [object](#_Trm00173) creation expression that includes an [object](#_Trm00173) initializer or collection initializer consists of first processing the [instance](#_Trm00172) constructor and then processing the member or element initializations specified by the [object](#_Trm00173) initializer ([§7.6.10.2](#_Toc00272)) or collection initializer ([§7.6.10.3](#_Toc00273)).

If any of the [arguments](#_Trm00062) in the optional [argument\_list](#_Grm00034) has the [compile-time type](#_Trm00074) dynamic then the [object\_creation\_expression](#_Grm00044) is dynamically bound ([§7.2.2](#_Toc00209)) and the following rules are applied at run-time using the [run-time type](#_Trm00073) of those [arguments](#_Trm00062) of the [argument\_list](#_Grm00034) that have the compile time type dynamic. However, the [object](#_Trm00173) creation undergoes a limited compile time check as described in [§7.5.4](#_Toc00249).

The [binding](#_Trm00210)-time processing of an [object\_creation\_expression](#_Grm00044) of the form new T(A), where T is a [class\_type](#_Grm00030) or a [value\_type](#_Grm00029) and A is an optional [argument\_list](#_Grm00034), consists of the following steps:

* If T is a [value\_type](#_Grm00029) and A is not present:
  + The [object\_creation\_expression](#_Grm00044) is a [default constructor](#_Trm00163) invocation. The result of the [object\_creation\_expression](#_Grm00044) is a [value](#_Trm00209) of type T, namely the [default value](#_Trm00164) for T as [defined](#_Trm00121) in [§4.1.1](#_Toc00092).
* Otherwise, if T is a [type\_parameter](#_Grm00032) and A is not present:
  + If no [value](#_Trm00209) type constraint or constructor constraint ([§10.1.5](#_Toc00399)) has been specified for T, a [binding](#_Trm00210)-time error occurs.
  + The result of the [object\_creation\_expression](#_Grm00044) is a [value](#_Trm00209) of the [run-time type](#_Trm00073) that the type parameter has been bound to, namely the result of invoking the [default constructor](#_Trm00163) of that type. The [run-time type](#_Trm00073) may be a reference type or a [value](#_Trm00209) type.
* Otherwise, if T is a [class\_type](#_Grm00030) or a [struct\_type](#_Grm00029):
  + If T is an abstract [class\_type](#_Grm00030), a compile-time error occurs.
  + The [instance](#_Trm00172) constructor to invoke is determined using the [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242). The set of candidate [instance](#_Trm00172) constructors consists of all [accessible](#_Trm00138) [instance](#_Trm00172) constructors declared in T which are applicable with respect to A ([§7.5.3.1](#_Toc00243)). If the set of candidate [instance](#_Trm00172) constructors is empty, or if a single best [instance](#_Trm00172) constructor cannot be identified, a [binding](#_Trm00210)-time error occurs.
  + The result of the [object\_creation\_expression](#_Grm00044) is a [value](#_Trm00209) of type T, namely the [value](#_Trm00209) produced by invoking the [instance](#_Trm00172) constructor determined in the step above.
* Otherwise, the [object\_creation\_expression](#_Grm00044) is invalid, and a [binding](#_Trm00210)-time error occurs.

Even if the [object\_creation\_expression](#_Grm00044) is dynamically bound, the [compile-time type](#_Trm00074) is still T.

The run-time processing of an [object\_creation\_expression](#_Grm00044) of the form new T(A), where T is [class\_type](#_Grm00030) or a [struct\_type](#_Grm00029) and A is an optional [argument\_list](#_Grm00034), consists of the following steps:

* If T is a [class\_type](#_Grm00030):
  + A new [instance](#_Trm00172) of class T is allocated. If there is not enough memory available to allocate the new [instance](#_Trm00172), a System.OutOfMemoryException is thrown and no further steps are executed.
  + All fields of the new [instance](#_Trm00172) are initialized to their [default value](#_Trm00164)s ([§5.2](#_Toc00131)).
  + The [instance](#_Trm00172) constructor is invoked according to the rules of function member invocation ([§7.5.4](#_Toc00249)). A reference to the newly allocated [instance](#_Trm00172) is automatically passed to the [instance](#_Trm00172) constructor and the [instance](#_Trm00172) can be accessed from within that constructor as this.
* If T is a [struct\_type](#_Grm00029):
  + An [instance](#_Trm00172) of type T is created by allocating a temporary [local variable](#_Trm00193). Since an [instance](#_Trm00172) constructor of a [struct\_type](#_Grm00029) is required to definitely assign a [value](#_Trm00209) to each field of the [instance](#_Trm00172) being created, no initialization of the temporary variable is necessary.
  + The [instance](#_Trm00172) constructor is invoked according to the rules of function member invocation ([§7.5.4](#_Toc00249)). A reference to the newly allocated [instance](#_Trm00172) is automatically passed to the [instance](#_Trm00172) constructor and the [instance](#_Trm00172) can be accessed from within that constructor as this.

#### Object initializers

An ***object initializer*** specifies [value](#_Trm00209)s for zero or more fields or properties of an [object](#_Trm00173).

object\_initializer:  
 | '{' member\_initializer\_list? '}'  
 | '{' member\_initializer\_list ',' '}'  
 ;  
  
member\_initializer\_list:  
 | member\_initializer ( ',' member\_initializer )\*  
 ;  
  
member\_initializer:  
 | identifier '=' initializer\_value  
 ;  
  
initializer\_value:  
 | expression  
 | object\_or\_collection\_initializer  
 ;

An [object](#_Trm00173) initializer consists of a sequence of member initializers, enclosed by { and } tokens and separated by commas. Each member initializer must name an [accessible](#_Trm00138) field or property of the [object](#_Trm00173) being initialized, followed by an equals sign and an expression or an [object](#_Trm00173) initializer or collection initializer. It is an error for an [object](#_Trm00173) initializer to include more than one member initializer for the same field or property. It is not possible for the [object](#_Trm00173) initializer to refer to the newly created [object](#_Trm00173) it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment ([§7.17.1](#_Toc00342)) to the field or property.

A member initializer that specifies an [object](#_Trm00173) initializer after the equals sign is a ***nested object initializer***, i.e. an initialization of an embedded [object](#_Trm00173). Instead of assigning a new [value](#_Trm00209) to the field or property, the assignments in the [nested](#_Trm00143) [object](#_Trm00173) initializer are treated as assignments to [members](#_Trm00012) of the field or property. Nested [object](#_Trm00173) initializers cannot be applied to properties with a [value](#_Trm00209) type, or to [read-only fields](#_Trm00055) with a [value](#_Trm00209) type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the field or property, the [elements](#_Trm00094) given in the initializer are added to the collection referenced by the field or property. The field or property must be of a collection type that satisfies the requirements specified in [§7.6.10.3](#_Toc00273).

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](#_Trm00172) 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 in[visible](#_Trm00152) and in[accessible](#_Trm00138) 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](#_Trm00172) 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](#_Trm00031) that are otherwise in[visible](#_Trm00152) and in[accessible](#_Trm00138).

If Rectangle's constructor allocates the two embedded Point [instance](#_Trm00172)s

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 [instance](#_Trm00172)s instead of assigning new [instance](#_Trm00172)s:

Rectangle r = new Rectangle {  
 P1 = { X = 0, Y = 1 },  
 P2 = { X = 2, Y = 3 }  
};

which has the same effect as

Rectangle \_\_r = new Rectangle();  
\_\_r.P1.X = 0;  
\_\_r.P1.Y = 1;  
\_\_r.P2.X = 2;  
\_\_r.P2.Y = 3;  
Rectangle r = \_\_r;

#### Collection initializers

A collection initializer specifies the [elements](#_Trm00094) 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](#_Trm00173) 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](#_Grm00067) production is [defined](#_Trm00121) in [§7.18](#_Toc00345).

The following is an example of an [object](#_Trm00173) 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](#_Trm00173) 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](#_Trm00056) on the target [object](#_Trm00173) with the expression list of the element initializer as argument list, applying normal [overload resolution](#_Trm00078) for each invocation. Thus, the collection [object](#_Trm00173) must contain an applicable Add [method](#_Trm00056) 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](#_Trm00031) that are otherwise in[visible](#_Trm00152) and in[accessible](#_Trm00138).

#### Array creation expressions

An [array\_creation\_expression](#_Grm00047) is used to create a new [instance](#_Trm00172) of an [array\_type](#_Grm00030).

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](#_Trm00093) creation expression of the first form allocates an [array](#_Trm00093) [instance](#_Trm00172) of the type that results from deleting each of the individual expressions from the expression list. For example, the [array](#_Trm00093) creation expression new int[10,20] produces an [array](#_Trm00093) [instance](#_Trm00172) of type int[,], and the [array](#_Trm00093) creation expression new int[10][,] produces an [array](#_Trm00093) of type int[][,]. Each expression in the expression list must be of type int, uint, long, or ulong, or [implicit](#_Trm00197)ly convertible to one or more of these [types](#_Trm00011). The [value](#_Trm00209) of each expression determines the [length](#_Trm00096) of the corresponding dimension in the newly allocated [array](#_Trm00093) [instance](#_Trm00172). Since the [length](#_Trm00096) of an [array](#_Trm00093) dimension must be nonnegative, it is a compile-time error to have a [constant\_expression](#_Grm00068) with a negative [value](#_Trm00209) in the expression list.

Except in an unsafe context ([§18.1](#_Toc00591)), the layout of [array](#_Trm00093)s is unspecified.

If an [array](#_Trm00093) creation expression of the first form includes an [array](#_Trm00093) initializer, each expression in the expression list must be a constant and the [rank](#_Trm00099) and dimension [length](#_Trm00096)s specified by the expression list must match those of the [array](#_Trm00093) initializer.

In an [array](#_Trm00093) creation expression of the second or third form, the [rank](#_Trm00099) of the specified [array](#_Trm00093) type or [rank](#_Trm00099) specifier must match that of the [array](#_Trm00093) initializer. The individual dimension [length](#_Trm00096)s are inferred from the number of [elements](#_Trm00094) in each of the corresponding nesting levels of the [array](#_Trm00093) 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](#_Trm00093) 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](#_Trm00095) of the [array](#_Trm00093) is not [explicit](#_Trm00198)ly given, but determined as the best common type ([§7.5.2.14](#_Toc00241)) of the set of expressions in the [array](#_Trm00093) initializer. For a multidimensional [array](#_Trm00093), i.e., one where the [rank\_specifier](#_Grm00030) contains at least one comma, this set comprises all [expression](#_Grm00067)s found in [nested](#_Trm00143) [array\_initializer](#_Grm00132)s.

Array initializers are described further in [§12.6](#_Toc00526).

The result of evaluating an [array](#_Trm00093) creation expression is classified as a [value](#_Trm00209), namely a reference to the newly allocated [array](#_Trm00093) [instance](#_Trm00172). The run-time processing of an [array](#_Trm00093) creation expression consists of the following steps:

* The dimension [length](#_Trm00096) expressions of the [expression\_list](#_Grm00046) are evaluated in order, from left to right. Following evaluation of each expression, an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) to one of the following [types](#_Trm00011) is performed: int, uint, long, ulong. The first type in this list for which an [implicit](#_Trm00197) [conversion](#_Trm00196) exists is chosen. If evaluation of an expression or the subsequent [implicit](#_Trm00197) [conversion](#_Trm00196) causes an exception, then no further expressions are evaluated and no further steps are executed.
* The computed [value](#_Trm00209)s for the dimension [length](#_Trm00096)s are validated as follows. If one or more of the [value](#_Trm00209)s are less than zero, a System.OverflowException is thrown and no further steps are executed.
* An [array](#_Trm00093) [instance](#_Trm00172) with the given dimension [length](#_Trm00096)s is allocated. If there is not enough memory available to allocate the new [instance](#_Trm00172), a System.OutOfMemoryException is thrown and no further steps are executed.
* All [elements](#_Trm00094) of the new [array](#_Trm00093) [instance](#_Trm00172) are initialized to their [default value](#_Trm00164)s ([§5.2](#_Toc00131)).
* If the [array](#_Trm00093) creation expression contains an [array](#_Trm00093) initializer, then each expression in the [array](#_Trm00093) initializer is evaluated and assigned to its corresponding [array](#_Trm00093) element. The evaluations and assignments are performed in the order the expressions are written in the [array](#_Trm00093) initializer—in other words, [elements](#_Trm00094) 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](#_Trm00093) element causes an exception, then no further [elements](#_Trm00094) are initialized (and the remaining [elements](#_Trm00094) will thus have their [default value](#_Trm00164)s).

An [array](#_Trm00093) creation expression permits instantiation of an [array](#_Trm00093) with [elements](#_Trm00094) of an [array](#_Trm00093) type, but the [elements](#_Trm00094) of such an [array](#_Trm00093) must be manually initialized. For example, the statement

int[][] a = new int[100][];

creates a single-dimensional [array](#_Trm00093) with 100 [elements](#_Trm00094) of type int[]. The initial [value](#_Trm00209) of each element is null. It is not possible for the same [array](#_Trm00093) creation expression to also instantiate the sub-[array](#_Trm00093)s, and the statement

int[][] a = new int[100][5]; // Error

results in a compile-time error. Instantiation of the sub-[array](#_Trm00093)s 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](#_Trm00093) of [array](#_Trm00093)s has a "rectangular" shape, that is when the sub-[array](#_Trm00093)s are all of the same [length](#_Trm00096), it is more efficient to use a multi-dimensional [array](#_Trm00093). In the example above, instantiation of the [array](#_Trm00093) of [array](#_Trm00093)s creates 101 [object](#_Trm00173)s—one outer [array](#_Trm00093) and 100 sub-[array](#_Trm00093)s. In contrast,

int[,] = new int[100, 5];

creates only a single [object](#_Trm00173), a two-dimensional [array](#_Trm00093), and accomplishes the allocation in a single statement.

The following are examples of [implicit](#_Trm00197)ly typed [array](#_Trm00093) 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 [implicit](#_Trm00197)ly convertible to the other, and so there is no best common type. An [explicit](#_Trm00198)ly typed [array](#_Trm00093) creation expression must be used in this case, for example specifying the type to be object[]. Alternatively, one of the [elements](#_Trm00094) can be cast to a common base type, which would then become the inferred [element type](#_Trm00095).

Implicitly typed [array](#_Trm00093) creation expressions can be combined with anonymous [object](#_Trm00173) initializers ([§7.6.10.6](#_Toc00276)) 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](#_Grm00048) is used to create a new [instance](#_Trm00172) of a [delegate\_type](#_Grm00030).

delegate\_creation\_expression:  
 | 'new' delegate\_type '(' expression ')'  
 ;

The argument of a delegate creation expression must be a [method](#_Trm00056) group, an anonymous function or a [value](#_Trm00209) of either the compile time type dynamic or a [delegate\_type](#_Grm00030). If the argument is a [method](#_Trm00056) group, it identifies the [method](#_Trm00056) and, for an [instance](#_Trm00172) [method](#_Trm00056), the [object](#_Trm00173) for which to create a delegate. If the argument is an anonymous function it directly defines the [parameters](#_Trm00059) and [method](#_Trm00056) body of the delegate target. If the argument is a [value](#_Trm00209) it identifies a delegate [instance](#_Trm00172) of which to create a copy.

If the [expression](#_Grm00067) has the [compile-time type](#_Trm00074) dynamic, the [delegate\_creation\_expression](#_Grm00048) is dynamically bound ([§7.2.2](#_Toc00209)), and the rules below are applied at run-time using the [run-time type](#_Trm00073) of the [expression](#_Grm00067). Otherwise the rules are applied at compile-time.

The [binding](#_Trm00210)-time processing of a [delegate\_creation\_expression](#_Grm00048) of the form new D(E), where D is a [delegate\_type](#_Grm00030) and E is an [expression](#_Grm00067), consists of the following steps:

* If E is a [method](#_Trm00056) group, the delegate creation expression is processed in the same way as a [method](#_Trm00056) group [conversion](#_Trm00196) ([§6.6](#_Toc00203)) 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](#_Trm00196) ([§6.5](#_Toc00199)) from E to D.
* If E is a [value](#_Trm00209), E must be compatible ([§15.1](#_Toc00558)) 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](#_Grm00048) of the form new D(E), where D is a [delegate\_type](#_Grm00030) and E is an [expression](#_Grm00067), consists of the following steps:

* If E is a [method](#_Trm00056) group, the delegate creation expression is evaluated as a [method](#_Trm00056) group [conversion](#_Trm00196) ([§6.6](#_Toc00203)) from E to D.
* If E is an anonymous function, the delegate creation is evaluated as an anonymous function [conversion](#_Trm00196) from E to D ([§6.5](#_Toc00199)).
* If E is a [value](#_Trm00209) of a [delegate\_type](#_Grm00030):
  + E is evaluated. If this evaluation causes an exception, no further steps are executed.
  + If the [value](#_Trm00209) of E is null, a System.NullReferenceException is thrown and no further steps are executed.
  + A new [instance](#_Trm00172) of the [delegate type](#_Trm00107) D is allocated. If there is not enough memory available to allocate the new [instance](#_Trm00172), a System.OutOfMemoryException is thrown and no further steps are executed.
  + The new delegate [instance](#_Trm00172) is initialized with the same invocation list as the delegate [instance](#_Trm00172) given by E.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When two delegates are combined or one is removed from another ([§15.1](#_Toc00558)), 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](#_Trm00087), user-[defined](#_Trm00121) [operator](#_Trm00090), [instance](#_Trm00172) constructor, [destructor](#_Trm00091), or [static constructor](#_Trm00081).

As described above, when a delegate is created from a [method](#_Trm00056) group, the formal parameter list and [return type](#_Trm00060) of the delegate determine which of the [overloaded](#_Trm00036) [method](#_Trm00056)s 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](#_Trm00056) because that [method](#_Trm00056) exactly matches the formal parameter list and [return type](#_Trm00060) of DoubleFunc. Had the second Square [method](#_Trm00056) not been present, a compile-time error would have occurred.

#### Anonymous [object](#_Trm00173) creation expressions

An [anonymous\_object\_creation\_expression](#_Grm00049) is used to create an [object](#_Trm00173) 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  
 | identifier '=' expression  
 ;

An anonymous [object](#_Trm00173) initializer declares an anonymous type and returns an [instance](#_Trm00172) of that type. An anonymous type is a nameless class type that inherits directly from object. The [members](#_Trm00012) of an anonymous type are a sequence of read-only properties inferred from the anonymous [object](#_Trm00173) initializer used to create an [instance](#_Trm00172) of the type. Specifically, an anonymous [object](#_Trm00173) 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](#_Grm00049) must have a type. Thus, it is a compile-time error for an expression in a [member\_declarator](#_Grm00049) 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](#_Trm00056) are automatically generated by the compiler and cannot be referenced in [program](#_Trm00109) text.

Within the same [program](#_Trm00109), two anonymous [object](#_Trm00173) initializers that specify a sequence of properties of the same names and [compile-time type](#_Trm00074)s in the same order will produce [instance](#_Trm00172)s 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 [method](#_Trm00056)s on anonymous [types](#_Trm00011) override the [method](#_Trm00056)s [inherited](#_Trm00136) from object, and are [defined](#_Trm00121) in terms of the Equals and GetHashcode of the properties, so that two [instance](#_Trm00172)s of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name ([§7.5.2](#_Toc00227)), a member access ([§7.5.4](#_Toc00249)) or a base access ([§7.6.8](#_Toc00268)). 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](#_Trm00246) the [identifier](#_Grm00007) selects both the [value](#_Trm00209) and the field or property to which the [value](#_Trm00209) is assigned. Intuitively, a [projection initializer](#_Trm00246) projects not just a [value](#_Trm00209), but also the name of the [value](#_Trm00209).

### The typeof [operator](#_Trm00090)

The typeof [operator](#_Trm00090) is used to obtain the System.Type [object](#_Trm00173) 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](#_Grm00050) consists of a typeof [keyword](#_Trm00117) followed by a parenthesized [type](#_Grm00028). The result of an expression of this form is the System.Type [object](#_Trm00173) for the indicated type. There is only one System.Type [object](#_Trm00173) for any given type. This means that for a type T, typeof(T) == typeof(T) is always true. The [type](#_Grm00028) cannot be dynamic.

The second form of [typeof\_expression](#_Grm00050) consists of a typeof [keyword](#_Trm00117) followed by a parenthesized [unbound\_type\_name](#_Grm00050). An [unbound\_type\_name](#_Grm00050) is very similar to a [type\_name](#_Grm00027) ([§3.8](#_Toc00086)) except that an [unbound\_type\_name](#_Grm00050) contains [generic\_dimension\_specifier](#_Grm00050)s where a [type\_name](#_Grm00027) contains [type\_argument\_list](#_Grm00031)s. When the operand of a [typeof\_expression](#_Grm00050) is a sequence of tokens that satisfies the grammars of both [unbound\_type\_name](#_Grm00050) and [type\_name](#_Grm00027), namely when it contains neither a [generic\_dimension\_specifier](#_Grm00050) nor a [type\_argument\_list](#_Grm00031), the sequence of tokens is considered to be a [type\_name](#_Grm00027). The meaning of an [unbound\_type\_name](#_Grm00050) is determined as follows:

* Convert the sequence of tokens to a [type\_name](#_Grm00027) by replacing each [generic\_dimension\_specifier](#_Grm00050) with a [type\_argument\_list](#_Grm00031) having the same number of commas and the [keyword](#_Trm00117) object as each [type\_argument](#_Grm00031).
* Evaluate the resulting [type\_name](#_Grm00027), while ignoring all type parameter constraints.
* The [unbound\_type\_name](#_Grm00050) resolves to the [unbound generic type](#_Trm00176) associated with the resulting [constructed type](#_Trm00178) ([§4.4.3](#_Toc00116)).

The result of the [typeof\_expression](#_Grm00050) is the System.Type [object](#_Trm00173) for the resulting [unbound generic type](#_Trm00176).

The third form of [typeof\_expression](#_Grm00050) consists of a typeof [keyword](#_Trm00117) followed by a parenthesized void [keyword](#_Trm00117). The result of an expression of this form is the System.Type [object](#_Trm00173) that represents the absence of a type. The type [object](#_Trm00173) returned by typeof(void) is distinct from the type [object](#_Trm00173) returned for any type. This special type [object](#_Trm00173) is useful in class [libraries](#_Trm00015) that allow reflection onto [method](#_Trm00056)s in the language, where those [method](#_Trm00056)s wish to have a way to represent the [return type](#_Trm00060) of any [method](#_Trm00056), including void [method](#_Trm00056)s, with an [instance](#_Trm00172) of System.Type.

The typeof [operator](#_Trm00090) can be used on a type parameter. The result is the System.Type [object](#_Trm00173) for the [run-time type](#_Trm00073) that was bound to the type parameter. The typeof [operator](#_Trm00090) can also be used on a [constructed type](#_Trm00178) or an [unbound generic type](#_Trm00176) ([§4.4.3](#_Toc00116)). The System.Type [object](#_Trm00173) for an [unbound generic type](#_Trm00176) is not the same as the System.Type [object](#_Trm00173) of the [instance](#_Trm00172) type. The [instance](#_Trm00172) type is always a closed [constructed type](#_Trm00178) at run-time so its System.Type [object](#_Trm00173) depends on the [run-time type](#_Trm00073) [arguments](#_Trm00062) in use, while the [unbound generic type](#_Trm00176) has no type [arguments](#_Trm00062).

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 [operator](#_Trm00090)s

The checked and unchecked [operator](#_Trm00090)s are used to control the ***overflow checking context*** for integral-type arithmetic operations and [conversion](#_Trm00196)s.

checked\_expression:  
 | 'checked' '(' expression ')'  
 ;  
  
unchecked\_expression:  
 | 'unchecked' '(' expression ')'  
 ;

The checked [operator](#_Trm00090) evaluates the contained expression in a checked context, and the unchecked [operator](#_Trm00090) evaluates the contained expression in an unchecked context. A [checked\_expression](#_Grm00051) or [unchecked\_expression](#_Grm00051) corresponds exactly to a [parenthesized\_expression](#_Grm00037) ([§7.6.3](#_Toc00256)), except that the contained expression is evaluated in the given [overflow checking context](#_Trm00247).

The [overflow checking context](#_Trm00247) can also be controlled through the checked and unchecked [statements](#_Trm00037) ([§8.11](#_Toc00373)).

The following operations are affected by the [overflow checking context](#_Trm00247) established by the checked and unchecked [operator](#_Trm00090)s and [statements](#_Trm00037):

* The pre[defined](#_Trm00121) ++ and -- unary [operator](#_Trm00090)s ([§7.6.9](#_Toc00269) and [§7.7.5](#_Toc00286)), when the operand is of an integral type.
* The pre[defined](#_Trm00121) - unary [operator](#_Trm00090) ([§7.7.2](#_Toc00283)), when the operand is of an integral type.
* The pre[defined](#_Trm00121) +, -, \*, and / binary [operator](#_Trm00090)s ([§7.8](#_Toc00292)), when both [operands](#_Trm00033) are of integral [types](#_Trm00011).
* Explicit numeric [conversion](#_Trm00196)s ([§6.2.1](#_Toc00182)) from one integral type to another integral type, or from float or double to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

* In a checked context, if the operation is a constant expression ([§7.19](#_Toc00346)), 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 [operator](#_Trm00090)s or [statements](#_Trm00037), the default [overflow checking context](#_Trm00247) 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](#_Trm00247) is always checked. Unless a constant expression is [explicit](#_Trm00198)ly 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](#_Trm00056) throws a System.OverflowException, and the G [method](#_Trm00056) returns -727379968 (the lower 32 bits of the out-of-range result). The behavior of the H [method](#_Trm00056) depends on the default [overflow checking context](#_Trm00247) 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 [operator](#_Trm00090)s only affect the [overflow checking context](#_Trm00247) for those operations that are textually contained within the "(" and ")" tokens. The [operator](#_Trm00090)s have no effect on [function members](#_Trm00079) 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](#_Trm00247).

The unchecked [operator](#_Trm00090) is convenient when writing constants of the signed integral [types](#_Trm00011) 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](#_Trm00090), the casts to int would produce compile-time errors.

The checked and unchecked [operator](#_Trm00090)s and [statements](#_Trm00037) allow [program](#_Trm00109)mers to control certain aspects of some numeric calculations. However, the behavior of some numeric [operator](#_Trm00090)s depends on their [operands](#_Trm00033)' data [types](#_Trm00011). For example, multiplying two decimals always results in an exception on overflow even within an [explicit](#_Trm00198)ly unchecked construct. Similarly, multiplying two floats never results in an exception on overflow even within an [explicit](#_Trm00198)ly checked construct. In addition, other [operator](#_Trm00090)s are never affected by the mode of checking, whether default or [explicit](#_Trm00198).

### Default [value](#_Trm00209) expressions

A [default value](#_Trm00164) expression is used to obtain the [default value](#_Trm00164) ([§5.2](#_Toc00131)) of a type. Typically a [default value](#_Trm00164) expression is used for type [parameters](#_Trm00059), since it may not be known if the type parameter is a [value](#_Trm00209) type or a reference type. (No [conversion](#_Trm00196) exists from the null [literal](#_Trm00118) to a type parameter unless the type parameter is known to be a reference type.)

default\_value\_expression:  
 | 'default' '(' type ')'  
 ;

If the [type](#_Grm00028) in a [default\_value\_expression](#_Grm00052) evaluates at run-time to a reference type, the result is null converted to that type. If the [type](#_Grm00028) in a [default\_value\_expression](#_Grm00052) evaluates at run-time to a [value](#_Trm00209) type, the result is the [value\_type](#_Grm00029)'s [default value](#_Trm00164) ([§4.1.2](#_Toc00093)).

A [default\_value\_expression](#_Grm00052) is a constant expression ([§7.19](#_Toc00346)) if the type is a reference type or a type parameter that is known to be a reference type ([§10.1.5](#_Toc00399)). In addition, a [default\_value\_expression](#_Grm00052) is a constant expression if the type is one of the following [value](#_Trm00209) [types](#_Trm00011): sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, or any enumeration type.

### Anonymous [method](#_Trm00056) expressions

An [anonymous\_method\_expression](#_Grm00064) is one of two ways of defining an anonymous function. These are further described in [§7.15](#_Toc00321).

## Unary [operator](#_Trm00090)s

The +, -, !, ~, ++, --, cast, and await [operator](#_Trm00090)s are called the unary [operator](#_Trm00090)s.

unary\_expression:  
 | primary\_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](#_Grm00053) has the [compile-time type](#_Trm00074) dynamic, it is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the [unary\_expression](#_Grm00053) is dynamic, and the resolution described below will take place at run-time using the [run-time type](#_Trm00073) of the operand.

### Unary plus [operator](#_Trm00090)

For an operation of the form +x, unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is applied to select a specific [operator](#_Trm00090) implementation. The operand is converted to the parameter type of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090). The pre[defined](#_Trm00121) unary plus [operator](#_Trm00090)s 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 [operator](#_Trm00090)s, the result is simply the [value](#_Trm00209) of the operand.

### Unary minus [operator](#_Trm00090)

For an operation of the form -x, unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is applied to select a specific [operator](#_Trm00090) implementation. The operand is converted to the parameter type of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090). The pre[defined](#_Trm00121) negation [operator](#_Trm00090)s are:

* Integer negation:

int operator -(int x);  
long operator -(long x);

The result is computed by subtracting x from zero. If the [value](#_Trm00209) of of x is the smallest representable [value](#_Trm00209) 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](#_Trm00209) of the operand and the overflow is not reported.

If the operand of the negation [operator](#_Trm00090) 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](#_Trm00209) -2147483648 (-2^31) to be written as a decimal integer [literal](#_Trm00118) ([§2.4.4.2](#_Toc00048)).

If the operand of the negation [operator](#_Trm00090) is of type ulong, a compile-time error occurs. An exception is the rule that permits the long [value](#_Trm00209) -9223372036854775808 (-2^63) to be written as a decimal integer [literal](#_Trm00118) ([§2.4.4.2](#_Toc00048)).

* Floating-point negation:

float operator -(float x);  
double operator -(double x);

The result is the [value](#_Trm00209) 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](#_Trm00090) of type System.Decimal.

### Logical negation [operator](#_Trm00090)

For an operation of the form !x, unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is applied to select a specific [operator](#_Trm00090) implementation. The operand is converted to the parameter type of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090). Only one pre[defined](#_Trm00121) logical negation [operator](#_Trm00090) exists:

bool operator !(bool x);

This [operator](#_Trm00090) 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](#_Trm00090)

For an operation of the form ~x, unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is applied to select a specific [operator](#_Trm00090) implementation. The operand is converted to the parameter type of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090). The pre[defined](#_Trm00121) bitwise complement [operator](#_Trm00090)s are:

int operator ~(int x);  
uint operator ~(uint x);  
long operator ~(long x);  
ulong operator ~(ulong x);

For each of these [operator](#_Trm00090)s, the result of the operation is the bitwise complement of x.

Every enumeration type E [implicit](#_Trm00197)ly provides the following bitwise complement [operator](#_Trm00090):

E operator ~(E x);

The result of evaluating ~x, where x is an expression of an enumeration type E with an [underlying type](#_Trm00106) U, is exactly the same as evaluating (E)(~(U)x), except that the [conversion](#_Trm00196) to E is always performed as if in an unchecked context ([§7.6.12](#_Toc00278)).

### Prefix increment and decrement [operator](#_Trm00090)s

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](#_Trm00087) access. The result of the operation is a [value](#_Trm00209) of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or [indexer](#_Trm00087) access, the property or [indexer](#_Trm00087) must have both a get and a set accessor. If this is not the case, a [binding](#_Trm00210)-time error occurs.

Unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is applied to select a specific [operator](#_Trm00090) implementation. Pre[defined](#_Trm00121) ++ and -- [operator](#_Trm00090)s exist for the following [types](#_Trm00011): sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any [enum type](#_Trm00105). The pre[defined](#_Trm00121) ++ [operator](#_Trm00090)s return the [value](#_Trm00209) produced by adding 1 to the operand, and the pre[defined](#_Trm00121) -- [operator](#_Trm00090)s return the [value](#_Trm00209) 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](#_Trm00105), 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](#_Trm00090) is invoked with the [value](#_Trm00209) of x as its argument.
  + The [value](#_Trm00209) returned by the [operator](#_Trm00090) is stored in the location given by the evaluation of x.
  + The [value](#_Trm00209) returned by the [operator](#_Trm00090) becomes the result of the operation.
* If x is classified as a property or [indexer](#_Trm00087) access:
  + The [instance](#_Trm00172) expression (if x is not static) and the argument list (if x is an [indexer](#_Trm00087) 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](#_Trm00090) is invoked with the [value](#_Trm00209) returned by the get accessor as its argument.
  + The set accessor of x is invoked with the [value](#_Trm00209) returned by the [operator](#_Trm00090) as its value argument.
  + The [value](#_Trm00209) returned by the [operator](#_Trm00090) becomes the result of the operation.

The ++ and -- [operator](#_Trm00090)s also support postfix notation ([§7.6.9](#_Toc00269)). Typically, the result of x++ or x-- is the [value](#_Trm00209) of x before the operation, whereas the result of ++x or --x is the [value](#_Trm00209) of x after the operation. In either case, x itself has the same [value](#_Trm00209) 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](#_Trm00090) implementations for the two notations.

### Cast expressions

A [cast\_expression](#_Grm00055) is used to [explicit](#_Trm00198)ly convert an expression to a given type.

cast\_expression:  
 | '(' type ')' unary\_expression  
 ;

A [cast\_expression](#_Grm00055) of the form (T)E, where T is a [type](#_Grm00028) and E is a [unary\_expression](#_Grm00053), performs an [explicit](#_Trm00198) [conversion](#_Trm00196) ([§6.2](#_Toc00181)) of the [value](#_Trm00209) of E to type T. If no [explicit](#_Trm00198) [conversion](#_Trm00196) exists from E to T, a [binding](#_Trm00210)-time error occurs. Otherwise, the result is the [value](#_Trm00209) produced by the [explicit](#_Trm00198) [conversion](#_Trm00196). The result is always classified as a [value](#_Trm00209), even if E denotes a variable.

The grammar for a [cast\_expression](#_Grm00055) leads to certain syntactic ambiguities. For example, the expression (x)-y could either be interpreted as a [cast\_expression](#_Grm00055) (a cast of -y to type x) or as an [additive\_expression](#_Grm00057) combined with a [parenthesized\_expression](#_Grm00037) (which computes the [value](#_Trm00209) x - y).

To resolve [cast\_expression](#_Grm00055) ambiguities, the following rule exists: A sequence of one or more [token](#_Grm00005)s ([§2.3.3](#_Toc00041)) enclosed in parentheses is considered the start of a [cast\_expression](#_Grm00055) only if at least one of the following are true:

* The sequence of tokens is correct grammar for a [type](#_Grm00028), but not for an [expression](#_Grm00067).
* The sequence of tokens is correct grammar for a [type](#_Grm00028), and the token immediately following the closing parentheses is the token "~", the token "!", the token "(", an [identifier](#_Grm00007) ([§2.4.1](#_Toc00043)), a [literal](#_Grm00009) ([§2.4.4](#_Toc00046)), or any [keyword](#_Grm00008) ([§2.4.3](#_Toc00045)) 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](#_Grm00055)s, but (x)-y is not, even if x identifies a type. However, if x is a [keyword](#_Trm00117) that identifies a pre[defined](#_Trm00121) type (such as int), then all four forms are [cast\_expression](#_Grm00055)s (because such a [keyword](#_Trm00117) could not possibly be an expression by itself).

### Await expressions

The await [operator](#_Trm00090) 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](#_Grm00056) is only allowed in the body of an async function ([§10.14](#_Toc00483)). Within the nearest enclosing async function, an [await\_expression](#_Grm00056) may not occur in these places:

* Inside a [nested](#_Trm00143) (non-async) anonymous function
* In a catch or finally [block](#_Trm00038) of a [try\_statement](#_Grm00093)
* Inside the [block](#_Trm00038) of a [lock\_statement](#_Grm00095)
* In an unsafe context

Note that an [await\_expression](#_Grm00056) cannot occur in most places within a [query\_expression](#_Grm00065), 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](#_Grm00056) is called the ***task***. It represents an asynchronous operation that may or may not be complete at the time the [await\_expression](#_Grm00056) is evaluated. The purpose of the await [operator](#_Trm00090) is to suspend execution of the enclosing async function until the awaited [task](#_Trm00248) is complete, and then obtain its outcome.

#### Awaitable expressions

The [task](#_Trm00248) of an await expression is required to be ***awaitable***. An expression t is [awaitable](#_Trm00249) if one of the following holds:

* t is of compile time type dynamic
* t has an [accessible](#_Trm00138) [instance](#_Trm00172) or extension [method](#_Trm00056) called GetAwaiter with no [parameters](#_Trm00059) and no type [parameters](#_Trm00059), and a [return type](#_Trm00060) A for which all of the following hold:
  + A implements the [interface](#_Trm00102) System.Runtime.CompilerServices.INotifyCompletion (hereafter known as INotifyCompletion for brevity)
  + A has an [accessible](#_Trm00138), readable [instance](#_Trm00172) property IsCompleted of type bool
  + A has an [accessible](#_Trm00138) [instance](#_Trm00172) [method](#_Trm00056) GetResult with no [parameters](#_Trm00059) and no type [parameters](#_Trm00059)

The purpose of the GetAwaiter [method](#_Trm00056) is to obtain an ***awaiter*** for the [task](#_Trm00248). The type A is called the ***awaiter type*** for the await expression.

The purpose of the IsCompleted property is to determine if the [task](#_Trm00248) is already complete. If so, there is no need to suspend evaluation.

The purpose of the INotifyCompletion.OnCompleted [method](#_Trm00056) is to sign up a "continuation" to the [task](#_Trm00248); i.e. a delegate (of type System.Action) that will be invoked once the [task](#_Trm00248) is complete.

The purpose of the GetResult [method](#_Trm00056) is to obtain the outcome of the [task](#_Trm00248) once it is complete. This outcome may be successful completion, possibly with a result [value](#_Trm00209), or it may be an exception which is thrown by the GetResult [method](#_Trm00056).

#### Classification of await expressions

The expression await t is classified the same way as the expression (t).GetAwaiter().GetResult(). Thus, if the [return type](#_Trm00060) of GetResult is void, the [await\_expression](#_Grm00056) is classified as nothing. If it has a non-void [return type](#_Trm00060) T, the [await\_expression](#_Grm00056) is classified as a [value](#_Trm00209) of type T.

#### Runtime evaluation of await expressions

At runtime, the expression await t is evaluated as follows:

* An [awaiter](#_Trm00250) 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](#_Trm00102) System.Runtime.CompilerServices.ICriticalNotifyCompletion (hereafter known as ICriticalNotifyCompletion for brevity). This check is done at [binding](#_Trm00210) time; i.e. at runtime if a has the compile time type dynamic, and at compile time otherwise. Let r denote the resumption delegate ([§10.14](#_Toc00483)):
  + 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](#_Trm00209), that [value](#_Trm00209) is the result of the [await\_expression](#_Grm00056). Otherwise the result is nothing.

An [awaiter](#_Trm00250)'s implementation of the [interface](#_Trm00102) [method](#_Trm00056)s 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 un[defined](#_Trm00121).

## Arithmetic [operator](#_Trm00090)s

The \*, /, %, +, and - [operator](#_Trm00090)s are called the arithmetic [operator](#_Trm00090)s.

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](#_Trm00090) has the [compile-time type](#_Trm00074) dynamic, then the expression is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the expression is dynamic, and the resolution described below will take place at run-time using the [run-time type](#_Trm00073) of those [operands](#_Trm00033) that have the [compile-time type](#_Trm00074) dynamic.

### Multiplication [operator](#_Trm00090)

For an operation of the form x \* y, [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) multiplication [operator](#_Trm00090)s are listed below. The [operator](#_Trm00090)s 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 [value](#_Trm00209)s, zeros, infinities, and NaN's. In the table, x and y are positive finite [value](#_Trm00209)s. 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](#_Trm00209) is too large to represent in the decimal format, a System.OverflowException is thrown. If the result [value](#_Trm00209) 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](#_Trm00033).

Decimal multiplication is equivalent to using the multiplication [operator](#_Trm00090) of type System.Decimal.

### Division [operator](#_Trm00090)

For an operation of the form x / y, [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) division [operator](#_Trm00090)s are listed below. The [operator](#_Trm00090)s 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](#_Trm00209) of the right operand is zero, a System.DivideByZeroException is thrown.

The division rounds the result towards zero. Thus the absolute [value](#_Trm00209) of the result is the largest possible integer that is less than or equal to the absolute [value](#_Trm00209) of the quotient of the two [operands](#_Trm00033). The result is zero or positive when the two [operands](#_Trm00033) have the same sign and zero or negative when the two [operands](#_Trm00033) have opposite signs.

If the left operand is the smallest representable int or long [value](#_Trm00209) 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](#_Trm00121) as to whether a System.ArithmeticException (or a subclass thereof) is thrown or the overflow goes unreported with the resulting [value](#_Trm00209) 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 [value](#_Trm00209)s, zeros, infinities, and NaN's. In the table, x and y are positive finite [value](#_Trm00209)s. 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](#_Trm00209) of the right operand is zero, a System.DivideByZeroException is thrown. If the resulting [value](#_Trm00209) is too large to represent in the decimal format, a System.OverflowException is thrown. If the result [value](#_Trm00209) is too small to represent in the decimal format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representantable decimal [value](#_Trm00209) to the true mathematical result.

Decimal division is equivalent to using the division [operator](#_Trm00090) of type System.Decimal.

### Remainder [operator](#_Trm00090)

For an operation of the form x % y, [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) remainder [operator](#_Trm00090)s are listed below. The [operator](#_Trm00090)s 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](#_Trm00209) 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](#_Trm00209) 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 [value](#_Trm00209)s, zeros, infinities, and NaN's. In the table, x and y are positive finite [value](#_Trm00209)s. 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](#_Trm00056) of computing the remainder is analogous to that used for integer [operands](#_Trm00033), 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](#_Trm00209) 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](#_Trm00033), 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](#_Trm00090) of type System.Decimal.

### Addition [operator](#_Trm00090)

For an operation of the form x + y, [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) addition [operator](#_Trm00090)s are listed below. For numeric and enumeration [types](#_Trm00011), the pre[defined](#_Trm00121) addition [operator](#_Trm00090)s compute the sum of the two [operands](#_Trm00033). When one or both [operands](#_Trm00033) are of type string, the pre[defined](#_Trm00121) addition [operator](#_Trm00090)s concatenate the string representation of the [operands](#_Trm00033).

* 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 [value](#_Trm00209)s, zeros, infinities, and NaN's. In the table, x and y are nonzero finite [value](#_Trm00209)s, 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](#_Trm00209) 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](#_Trm00033).

Decimal addition is equivalent to using the addition [operator](#_Trm00090) of type System.Decimal.

* Enumeration addition. Every enumeration type [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) [operator](#_Trm00090)s, where E is the [enum type](#_Trm00105), and U is the [underlying type](#_Trm00106) of E:

E operator +(E x, U y);  
E operator +(U x, E y);

At run-time these [operator](#_Trm00090)s 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](#_Trm00090) 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](#_Trm00056) [inherited](#_Trm00136) 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](#_Trm00090) is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation [operator](#_Trm00090) never returns a null [value](#_Trm00209). A System.OutOfMemoryException may be thrown if there is not enough memory available to allocate the resulting string.

* Delegate combination. Every [delegate type](#_Trm00107) [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) [operator](#_Trm00090), where D is the [delegate type](#_Trm00107):

D operator +(D x, D y);

The binary + [operator](#_Trm00090) performs delegate combination when both [operands](#_Trm00033) are of some [delegate type](#_Trm00107) D. (If the [operands](#_Trm00033) have different [delegate type](#_Trm00107)s, a [binding](#_Trm00210)-time error occurs.) If the first operand is null, the result of the operation is the [value](#_Trm00209) 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](#_Trm00209) of the first operand. Otherwise, the result of the operation is a new delegate [instance](#_Trm00172) that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see [§7.8.5](#_Toc00297) and [§15.4](#_Toc00561). Since System.Delegate is not a [delegate type](#_Trm00107), operator + is not [defined](#_Trm00121) for it.

### Subtraction [operator](#_Trm00090)

For an operation of the form x - y, [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) subtraction [operator](#_Trm00090)s are listed below. The [operator](#_Trm00090)s 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 [value](#_Trm00209)s, zeros, infinities, and NaNs. In the table, x and y are nonzero finite [value](#_Trm00209)s, 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](#_Trm00209) 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](#_Trm00033).

Decimal subtraction is equivalent to using the subtraction [operator](#_Trm00090) of type System.Decimal.

* Enumeration subtraction. Every enumeration type [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) [operator](#_Trm00090), where E is the [enum type](#_Trm00105), and U is the [underlying type](#_Trm00106) of E:

U operator -(E x, E y);

This [operator](#_Trm00090) is evaluated exactly as (U)((U)x - (U)y). In other words, the [operator](#_Trm00090) computes the difference between the ordinal [value](#_Trm00209)s of x and y, and the type of the result is the [underlying type](#_Trm00106) of the enumeration.

E operator -(E x, U y);

This [operator](#_Trm00090) is evaluated exactly as (E)((U)x - y). In other words, the [operator](#_Trm00090) subtracts a [value](#_Trm00209) from the [underlying type](#_Trm00106) of the enumeration, yielding a [value](#_Trm00209) of the enumeration.

* Delegate removal. Every [delegate type](#_Trm00107) [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) [operator](#_Trm00090), where D is the [delegate type](#_Trm00107):

D operator -(D x, D y);

The binary - [operator](#_Trm00090) performs delegate removal when both [operands](#_Trm00033) are of some [delegate type](#_Trm00107) D. If the [operands](#_Trm00033) have different [delegate type](#_Trm00107)s, a [binding](#_Trm00210)-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](#_Trm00209) of the first operand. Otherwise, both [operands](#_Trm00033) represent invocation lists ([§15.1](#_Toc00558)) 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](#_Trm00090) ([§7.10.8](#_Toc00307)).) Otherwise, the result is the [value](#_Trm00209) of the left operand. Neither of the [operands](#_Trm00033)' 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 [operator](#_Trm00090)s

The << and >> [operator](#_Trm00090)s 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](#_Grm00058) has the [compile-time type](#_Trm00074) dynamic, then the expression is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the expression is dynamic, and the resolution described below will take place at run-time using the [run-time type](#_Trm00073) of those [operands](#_Trm00033) that have the [compile-time type](#_Trm00074) dynamic.

For an operation of the form x << count or x >> count, [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

When declaring an [overloaded](#_Trm00036) shift [operator](#_Trm00090), the type of the first operand must always be the class or struct containing the [operator](#_Trm00090) declaration, and the type of the second operand must always be int.

The pre[defined](#_Trm00121) shift [operator](#_Trm00090)s 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](#_Trm00090) 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](#_Trm00090) 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 pre[defined](#_Trm00121) [operator](#_Trm00090)s, 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 [operator](#_Trm00090)s simply return the [value](#_Trm00209) of x.

Shift operations never cause overflows and produce the same results in checked and unchecked contexts.

When the left operand of the >> [operator](#_Trm00090) is of a signed integral type, the [operator](#_Trm00090) performs an arithmetic shift right wherein the [value](#_Trm00209) 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](#_Trm00090) is of an unsigned integral type, the [operator](#_Trm00090) 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](#_Trm00198) 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 [operator](#_Trm00090)s

The ==, !=, <, >, <=, >=, is and as [operator](#_Trm00090)s are called the relational and type-testing [operator](#_Trm00090)s.

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](#_Trm00090) is described in [§7.10.10](#_Toc00309) and the as [operator](#_Trm00090) is described in [§7.10.11](#_Toc00310).

The ==, !=, <, >, <= and >= [operator](#_Trm00090)s are ***comparison operators***.

If an operand of a comparison [operator](#_Trm00090) has the [compile-time type](#_Trm00074) dynamic, then the expression is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the expression is dynamic, and the resolution described below will take place at run-time using the [run-time type](#_Trm00073) of those [operands](#_Trm00033) that have the [compile-time type](#_Trm00074) dynamic.

For an operation of the form x *op* y, where *op* is a comparison [operator](#_Trm00090), [overload resolution](#_Trm00078) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) [comparison operators](#_Trm00252) are described in the following sections. All pre[defined](#_Trm00121) [comparison operators](#_Trm00252) return a result of type bool, as described in the following table.

|  |  |
| --- | --- |
| **Operation** | **Result** |
| x == y | true if x is equal to y, false otherwise |
| x != y | true if x is not equal to y, false otherwise |
| x < y | true if x is less than y, false otherwise |
| x > y | true if x is greater than y, false otherwise |
| x <= y | true if x is less than or equal to y, false otherwise |
| x >= y | true if x is greater than or equal to y, false otherwise |

### Integer [comparison operators](#_Trm00252)

The pre[defined](#_Trm00121) integer [comparison operators](#_Trm00252) 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 [operator](#_Trm00090)s compares the numeric [value](#_Trm00209)s of the two integer [operands](#_Trm00033) and returns a bool [value](#_Trm00209) that indicates whether the particular relation is true or false.

### Floating-point [comparison operators](#_Trm00252)

The pre[defined](#_Trm00121) floating-point [comparison operators](#_Trm00252) 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 [operator](#_Trm00090)s compare the [operands](#_Trm00033) according to the rules of the IEEE 754 standard:

* If either operand is NaN, the result is false for all [operator](#_Trm00090)s except !=, for which the result is true. For any two [operands](#_Trm00033), x != y always produces the same result as !(x == y). However, when one or both [operands](#_Trm00033) are NaN, the <, >, <=, and >= [operator](#_Trm00090)s do not produce the same results as the logical negation of the opposite [operator](#_Trm00090). 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 [operator](#_Trm00090)s compare the [value](#_Trm00209)s of the two floating-point [operands](#_Trm00033) 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 [value](#_Trm00209)s 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 [value](#_Trm00209)s, but equal to another negative infinity.
  + A positive infinity is considered greater than all other [value](#_Trm00209)s, but equal to another positive infinity.

### Decimal [comparison operators](#_Trm00252)

The pre[defined](#_Trm00121) decimal [comparison operators](#_Trm00252) 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 [operator](#_Trm00090)s compares the numeric [value](#_Trm00209)s of the two decimal [operands](#_Trm00033) and returns a bool [value](#_Trm00209) that indicates whether the particular relation is true or false. Each decimal comparison is equivalent to using the corresponding relational or equality [operator](#_Trm00090) of type System.Decimal.

### Boolean equality [operator](#_Trm00090)s

The pre[defined](#_Trm00121) boolean equality [operator](#_Trm00090)s 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](#_Trm00033) are of type bool, the != [operator](#_Trm00090) produces the same result as the ^ [operator](#_Trm00090).

### Enumeration [comparison operators](#_Trm00252)

Every enumeration type [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) [comparison operators](#_Trm00252):

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](#_Trm00106) U, and op is one of the [comparison operators](#_Trm00252), is exactly the same as evaluating ((U)x) op ((U)y). In other words, the enumeration type [comparison operators](#_Trm00252) simply compare the underlying integral [value](#_Trm00209)s of the two [operands](#_Trm00033).

### Reference type equality [operator](#_Trm00090)s

The pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s are:

bool operator ==(object x, object y);  
bool operator !=(object x, object y);

The [operator](#_Trm00090)s return the result of comparing the two [references](#_Trm00160) for equality or non-equality.

Since the pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s accept [operands](#_Trm00033) of type object, they apply to all [types](#_Trm00011) that do not declare applicable operator == and operator != [members](#_Trm00012). Conversely, any applicable user-[defined](#_Trm00121) equality [operator](#_Trm00090)s effectively [hide](#_Trm00132) the pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s.

The pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s require one of the following:

* Both [operands](#_Trm00033) are a [value](#_Trm00209) of a type known to be a [reference\_type](#_Grm00030) or the [literal](#_Trm00118) null. Furthermore, an [explicit](#_Trm00198) reference [conversion](#_Trm00196) ([§6.2.4](#_Toc00185)) exists from the type of either operand to the type of the other operand.
* One operand is a [value](#_Trm00209) of type T where T is a [type\_parameter](#_Grm00032) and the other operand is the [literal](#_Trm00118) null. Furthermore T does not have the [value](#_Trm00209) type constraint.

Unless one of these conditions are true, a [binding](#_Trm00210)-time error occurs. Notable implications of these rules are:

* It is a [binding](#_Trm00210)-time error to use the pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s to compare two [references](#_Trm00160) that are known to be different at [binding](#_Trm00210)-time. For example, if the [binding](#_Trm00210)-time [types](#_Trm00011) of the [operands](#_Trm00033) are two [class types](#_Trm00024) A and B, and if neither A nor B derives from the other, then it would be impossible for the two [operands](#_Trm00033) to reference the same [object](#_Trm00173). Thus, the operation is considered a [binding](#_Trm00210)-time error.
* The pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s do not permit [value](#_Trm00209) type [operands](#_Trm00033) to be compared. Therefore, unless a struct type declares its own equality [operator](#_Trm00090)s, it is not possible to compare [value](#_Trm00209)s of that struct type.
* The pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090)s never cause [boxing](#_Trm00029) operations to occur for their [operands](#_Trm00033). It would be meaningless to perform such [boxing](#_Trm00029) operations, since [references](#_Trm00160) to the newly allocated boxed [instance](#_Trm00172)s would necessarily differ from all other [references](#_Trm00160).
* If an operand of a type parameter type T is compared to null, and the [run-time type](#_Trm00073) of T is a [value](#_Trm00209) 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](#_Trm00209) type, and the result is simply [defined](#_Trm00121) to be false when T is a [value](#_Trm00209) type.

For an operation of the form x == y or x != y, if any applicable operator == or operator != exists, the [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.4](#_Toc00215)) rules will select that [operator](#_Trm00090) instead of the pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090). However, it is always possible to select the pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090) by [explicit](#_Trm00198)ly casting one or both of the [operands](#_Trm00033) 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](#_Trm00031) refer to two distinct string [instance](#_Trm00172)s containing the same characters. The first comparison outputs True because the pre[defined](#_Trm00121) string equality [operator](#_Trm00090) ([§7.10.7](#_Toc00306)) is selected when both [operands](#_Trm00033) are of type string. The remaining comparisons all output False because the pre[defined](#_Trm00121) reference type equality [operator](#_Trm00090) is selected when one or both of the [operands](#_Trm00033) are of type object.

Note that the above technique is not meaningful for [value](#_Trm00209) [types](#_Trm00011). 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](#_Trm00160) to two separate [instance](#_Trm00172)s of boxed int [value](#_Trm00209)s.

### String equality [operator](#_Trm00090)s

The pre[defined](#_Trm00121) string equality [operator](#_Trm00090)s are:

bool operator ==(string x, string y);  
bool operator !=(string x, string y);

Two string [value](#_Trm00209)s are considered equal when one of the following is true:

* Both [value](#_Trm00209)s are null.
* Both [value](#_Trm00209)s are non-null [references](#_Trm00160) to string [instance](#_Trm00172)s that have identical [length](#_Trm00096)s and identical characters in each character position.

The string equality [operator](#_Trm00090)s compare string [value](#_Trm00209)s rather than string [references](#_Trm00160). When two separate string [instance](#_Trm00172)s contain the exact same sequence of characters, the [value](#_Trm00209)s of the strings are equal, but the [references](#_Trm00160) are different. As described in [§7.10.6](#_Toc00305), the reference type equality [operator](#_Trm00090)s can be used to compare string [references](#_Trm00160) instead of string [value](#_Trm00209)s.

### Delegate equality [operator](#_Trm00090)s

Every [delegate type](#_Trm00107) [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) [comparison operators](#_Trm00252):

bool operator ==(System.Delegate x, System.Delegate y);  
bool operator !=(System.Delegate x, System.Delegate y);

Two delegate [instance](#_Trm00172)s are considered equal as follows:

* If either of the delegate [instance](#_Trm00172)s is null, they are equal if and only if both are null.
* If the delegates have different [run-time type](#_Trm00073) they are never equal.
* If both of the delegate [instance](#_Trm00172)s have an invocation list ([§15.1](#_Toc00558)), those [instance](#_Trm00172)s are equal if and only if their invocation lists are the same [length](#_Trm00096), and each entry in one's invocation list is equal (as [defined](#_Trm00121) 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](#_Trm00056) then the entries are equal.
* If two invocation list entries both refer to the same non-static [method](#_Trm00056) on the same target [object](#_Trm00173) (as [defined](#_Trm00121) by the reference equality [operator](#_Trm00090)s) then the entries are equal.
* Invocation list entries produced from evaluation of semantically identical [anonymous\_method\_expression](#_Grm00064)s or [lambda\_expression](#_Grm00064)s with the same (possibly empty) set of captured outer variable [instance](#_Trm00172)s are permitted (but not required) to be equal.

### Equality [operator](#_Trm00090)s and null

The == and != [operator](#_Trm00090)s permit one operand to be a [value](#_Trm00209) of a nullable type and the other to be the null [literal](#_Trm00118), even if no pre[defined](#_Trm00121) or user-[defined](#_Trm00121) [operator](#_Trm00090) (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](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.4](#_Toc00215)) fails to find an applicable [operator](#_Trm00090), 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](#_Trm00090)

The is [operator](#_Trm00090) is used to dynamically check if the [run-time type](#_Trm00073) of an [object](#_Trm00173) 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](#_Trm00209) indicating whether E can successfully be converted to type T by a reference [conversion](#_Trm00196), a [boxing](#_Trm00029) [conversion](#_Trm00196), or an un[boxing](#_Trm00029) [conversion](#_Trm00196). The operation is evaluated as follows, after type [arguments](#_Trm00062) have been substituted for all type [parameters](#_Trm00059):

* If E is an anonymous function, a compile-time error occurs
* If E is a [method](#_Trm00056) group or the null [literal](#_Trm00118), of if the type of E is a reference type or a nullable type and the [value](#_Trm00209) 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](#_Trm00073) of the [instance](#_Trm00172) reference by E.
  + If the type of E is a nullable type, D is the [underlying type](#_Trm00106) of that nullable type.
  + If the type of E is a [non-nullable value type](#_Trm00169), 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](#_Trm00197) reference [conversion](#_Trm00196) from D to T exists, or if D is a [value](#_Trm00209) type and a [boxing](#_Trm00029) [conversion](#_Trm00196) from D to T exists.
  + If T is a nullable type, the result is true if D is the [underlying type](#_Trm00106) of T.
  + If T is a [non-nullable value type](#_Trm00169), the result is true if D and T are the same type.
  + Otherwise, the result is false.

Note that user [defined](#_Trm00121) [conversion](#_Trm00196)s, are not considered by the is [operator](#_Trm00090).

### The as [operator](#_Trm00090)

The as [operator](#_Trm00090) is used to [explicit](#_Trm00198)ly convert a [value](#_Trm00209) to a given reference type or nullable type. Unlike a cast expression ([§7.7.6](#_Toc00287)), the as [operator](#_Trm00090) never throws an exception. Instead, if the indicated [conversion](#_Trm00196) is not possible, the resulting [value](#_Trm00209) is null.

In an operation of the form E as T, E must be an expression and T must be a reference type, a type parameter known to be a reference type, or a nullable type. Furthermore, at least one of the following must be true, or otherwise a compile-time error occurs:

* An identity ([§6.1.1](#_Toc00169)), [implicit](#_Trm00197) nullable ([§6.1.4](#_Toc00172)), [implicit](#_Trm00197) reference ([§6.1.6](#_Toc00174)), [boxing](#_Trm00029) ([§6.1.7](#_Toc00175)), [explicit](#_Trm00198) nullable ([§6.2.3](#_Toc00184)), [explicit](#_Trm00198) reference ([§6.2.4](#_Toc00185)), or un[boxing](#_Trm00029) ([§6.2.5](#_Toc00186)) [conversion](#_Trm00196) exists from E to T.
* The type of E or T is an open type.
* E is the null [literal](#_Trm00118).

If the [compile-time type](#_Trm00074) 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](#_Trm00074) of E is dynamic, unlike the cast [operator](#_Trm00090) the as [operator](#_Trm00090) is not dynamically bound ([§7.2.2](#_Toc00209)). Therefore the expansion in this case is:

E is T ? (T)(object)(E) : (T)null

Note that some [conversion](#_Trm00196)s, such as user [defined](#_Trm00121) [conversion](#_Trm00196)s, are not possible with the as [operator](#_Trm00090) 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](#_Trm00090) in H is disallowed.

## Logical [operator](#_Trm00090)s

The &, ^, and | [operator](#_Trm00090)s are called the logical [operator](#_Trm00090)s.

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](#_Trm00090) has the [compile-time type](#_Trm00074) dynamic, then the expression is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the expression is dynamic, and the resolution described below will take place at run-time using the [run-time type](#_Trm00073) of those [operands](#_Trm00033) that have the [compile-time type](#_Trm00074) dynamic.

For an operation of the form x op y, where op is one of the logical [operator](#_Trm00090)s, [overload resolution](#_Trm00078) ([§7.3.4](#_Toc00215)) is applied to select a specific [operator](#_Trm00090) implementation. The [operands](#_Trm00033) are converted to the parameter [types](#_Trm00011) of the selected [operator](#_Trm00090), and the type of the result is the [return type](#_Trm00060) of the [operator](#_Trm00090).

The pre[defined](#_Trm00121) logical [operator](#_Trm00090)s are described in the following sections.

### Integer logical [operator](#_Trm00090)s

The pre[defined](#_Trm00121) integer logical [operator](#_Trm00090)s 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](#_Trm00090) computes the bitwise logical AND of the two [operands](#_Trm00033), the | [operator](#_Trm00090) computes the bitwise logical OR of the two [operands](#_Trm00033), and the ^ [operator](#_Trm00090) computes the bitwise logical exclusive OR of the two [operands](#_Trm00033). No overflows are possible from these operations.

### Enumeration logical [operator](#_Trm00090)s

Every enumeration type E [implicit](#_Trm00197)ly provides the following pre[defined](#_Trm00121) logical [operator](#_Trm00090)s:

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](#_Trm00106) U, and op is one of the logical [operator](#_Trm00090)s, is exactly the same as evaluating (E)((U)x op (U)y). In other words, the enumeration type logical [operator](#_Trm00090)s simply perform the logical operation on the [underlying type](#_Trm00106) of the two [operands](#_Trm00033).

### Boolean logical [operator](#_Trm00090)s

The pre[defined](#_Trm00121) boolean logical [operator](#_Trm00090)s 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](#_Trm00033) are of type bool, the ^ [operator](#_Trm00090) computes the same result as the != [operator](#_Trm00090).

### Nullable boolean logical [operator](#_Trm00090)s

The nullable boolean type bool? can represent three [value](#_Trm00209)s, true, false, and null, and is conceptually similar to the three-[value](#_Trm00209)d type used for boolean expressions in SQL. To ensure that the results produced by the & and | [operator](#_Trm00090)s for bool? [operands](#_Trm00033) are consistent with SQL's three-[value](#_Trm00209)d logic, the following pre[defined](#_Trm00121) [operator](#_Trm00090)s are provided:

bool? operator &(bool? x, bool? y);  
bool? operator |(bool? x, bool? y);

The following table lists the results produced by these [operator](#_Trm00090)s for all combinations of the [value](#_Trm00209)s 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 [operator](#_Trm00090)s

The && and || [operator](#_Trm00090)s are called the conditional logical [operator](#_Trm00090)s. They are also called the "short-circuiting" logical [operator](#_Trm00090)s.

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 || [operator](#_Trm00090)s are conditional versions of the & and | [operator](#_Trm00090)s:

* 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](#_Trm00090) has the [compile-time type](#_Trm00074) dynamic, then the expression is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the expression is dynamic, and the resolution described below will take place at run-time using the [run-time type](#_Trm00073) of those [operands](#_Trm00033) that have the [compile-time type](#_Trm00074) dynamic.

An operation of the form x && y or x || y is processed by applying [overload resolution](#_Trm00078) ([§7.3.4](#_Toc00215)) as if the operation was written x & y or x | y. Then,

* If [overload resolution](#_Trm00078) fails to find a single best [operator](#_Trm00090), or if [overload resolution](#_Trm00078) selects one of the pre[defined](#_Trm00121) integer logical [operator](#_Trm00090)s, a [binding](#_Trm00210)-time error occurs.
* Otherwise, if the selected [operator](#_Trm00090) is one of the pre[defined](#_Trm00121) boolean logical [operator](#_Trm00090)s ([§7.11.3](#_Toc00314)) or nullable boolean logical [operator](#_Trm00090)s ([§7.11.4](#_Toc00315)), the operation is processed as described in [§7.12.1](#_Toc00317).
* Otherwise, the selected [operator](#_Trm00090) is a user-[defined](#_Trm00121) [operator](#_Trm00090), and the operation is processed as described in [§7.12.2](#_Toc00318).

It is not possible to directly overload the conditional logical [operator](#_Trm00090)s. However, because the conditional logical [operator](#_Trm00090)s are evaluated in terms of the regular logical [operator](#_Trm00090)s, overloads of the regular logical [operator](#_Trm00090)s are, with certain restrictions, also considered overloads of the conditional logical [operator](#_Trm00090)s. This is described further in [§7.12.2](#_Toc00318).

### Boolean conditional logical [operator](#_Trm00090)s

When the [operands](#_Trm00033) of && or || are of type bool, or when the [operands](#_Trm00033) are of [types](#_Trm00011) that do not define an applicable operator & or operator |, but do define [implicit](#_Trm00197) [conversion](#_Trm00196)s 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](#_Trm00121) conditional logical [operator](#_Trm00090)s

When the [operands](#_Trm00033) of && or || are of [types](#_Trm00011) that declare an applicable user-[defined](#_Trm00121) operator & or operator |, both of the following must be true, where T is the type in which the selected [operator](#_Trm00090) is declared:

* The [return type](#_Trm00060) and the type of each parameter of the selected [operator](#_Trm00090) must be T. In other words, the [operator](#_Trm00090) must compute the logical AND or the logical OR of two [operands](#_Trm00033) of type T, and must return a result of type T.
* T must contain declarations of operator true and operator false.

A [binding](#_Trm00210)-time error occurs if either of these requirements is not satisfied. Otherwise, the && or || operation is evaluated by combining the user-[defined](#_Trm00121) operator true or operator false with the selected user-[defined](#_Trm00121) [operator](#_Trm00090):

* 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](#_Trm00209) previously computed for x. Otherwise, y is evaluated, and the selected operator & is invoked on the [value](#_Trm00209) previously computed for x and the [value](#_Trm00209) 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](#_Trm00209) previously computed for x. Otherwise, y is evaluated, and the selected operator | is invoked on the [value](#_Trm00209) previously computed for x and the [value](#_Trm00209) computed for y to produce the result of the operation.

In either of these operations, the expression given by x is only evaluated once, and the expression given by y is either not evaluated or evaluated exactly once.

For an example of a type that implements operator true and operator false, see [§11.4.2](#_Toc00517).

## The null coalescing [operator](#_Trm00090)

The ?? [operator](#_Trm00090) is called the null coalescing [operator](#_Trm00090).

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](#_Trm00090) is [right-associative](#_Trm00222), 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](#_Trm00033) that is non-null, or null if all [operands](#_Trm00033) are null.

The type of the expression a ?? b depends on which [implicit](#_Trm00197) [conversion](#_Trm00196)s are available on the [operands](#_Trm00033). 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](#_Trm00106) 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](#_Trm00174), 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](#_Trm00197) [conversion](#_Trm00196) 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](#_Trm00197) [conversion](#_Trm00196) 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](#_Trm00197) [conversion](#_Trm00196) 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](#_Trm00090)

The ?: [operator](#_Trm00090) is called the conditional [operator](#_Trm00090). It is at times also called the ternary [operator](#_Trm00090).

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](#_Trm00090) is [right-associative](#_Trm00222), 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](#_Trm00090) must be an expression that can be [implicit](#_Trm00197)ly 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](#_Trm00033), x and y, of the ?: [operator](#_Trm00090) control the type of the conditional expression.

* If x has type X and y has type Y then
  + If an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) exists from X to Y, but not from Y to X, then Y is the type of the conditional expression.
  + If an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) 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 [implicit](#_Trm00197)ly 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](#_Trm00209) of b is determined:
  + If an [implicit](#_Trm00197) [conversion](#_Trm00196) from the type of b to bool exists, then this [implicit](#_Trm00197) [conversion](#_Trm00196) is performed to produce a bool [value](#_Trm00209).
  + Otherwise, the operator true [defined](#_Trm00121) by the type of b is invoked to produce a bool [value](#_Trm00209).
* If the bool [value](#_Trm00209) 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](#_Trm00056) definition. An [anonymous function](#_Trm00253) does not have a [value](#_Trm00209) or type in and of itself, but is convertible to a compatible delegate or expression tree type. The evaluation of an [anonymous function](#_Trm00253) [conversion](#_Trm00196) depends on the [target type](#_Trm00203) of the [conversion](#_Trm00196): If it is a [delegate type](#_Trm00107), the [conversion](#_Trm00196) evaluates to a delegate [value](#_Trm00209) referencing the [method](#_Trm00056) which the [anonymous function](#_Trm00253) defines. If it is an expression tree type, the [conversion](#_Trm00196) evaluates to an expression tree which represents the structure of the [method](#_Trm00056) as an [object](#_Trm00173) structure.

For historical reasons there are two syntactic flavors of [anonymous function](#_Trm00253)s, namely [lambda\_expression](#_Grm00064)s and [anonymous\_method\_expression](#_Grm00064)s. For almost all purposes, [lambda\_expression](#_Grm00064)s are more concise and expressive than [anonymous\_method\_expression](#_Grm00064)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](#_Trm00090) has the same [precedence](#_Trm00035) as assignment (=) and is [right-associative](#_Trm00222).

An [anonymous function](#_Trm00253) with the async modifier is an async function and follows the rules described in [§10.14](#_Toc00483).

The [parameters](#_Trm00059) of an [anonymous function](#_Trm00253) in the form of a [lambda\_expression](#_Grm00064) can be [explicit](#_Trm00198)ly or [implicit](#_Trm00197)ly typed. In an [explicit](#_Trm00198)ly typed parameter list, the type of each parameter is [explicit](#_Trm00198)ly stated. In an [implicit](#_Trm00197)ly typed parameter list, the [types](#_Trm00011) of the [parameters](#_Trm00059) are inferred from the context in which the [anonymous function](#_Trm00253) occurs—specifically, when the [anonymous function](#_Trm00253) is converted to a compatible [delegate type](#_Trm00107) or expression tree type, that type provides the parameter [types](#_Trm00011) ([§6.5](#_Toc00199)).

In an [anonymous function](#_Trm00253) with a single, [implicit](#_Trm00197)ly typed parameter, the parentheses may be omitted from the parameter list. In other words, an [anonymous function](#_Trm00253) of the form

( param ) => expr

can be abbreviated to

param => expr

The parameter list of an [anonymous function](#_Trm00253) in the form of an [anonymous\_method\_expression](#_Grm00064) is optional. If given, the [parameters](#_Trm00059) must be [explicit](#_Trm00198)ly typed. If not, the [anonymous function](#_Trm00253) is convertible to a delegate with any parameter list not containing out [parameters](#_Trm00059).

A [block](#_Grm00071) body of an [anonymous function](#_Trm00253) is reachable ([§8.1](#_Toc00349)) unless the [anonymous function](#_Trm00253) occurs inside an unreachable statement.

Some examples of [anonymous function](#_Trm00253)s 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](#_Grm00064)s and [anonymous\_method\_expression](#_Grm00064)s is the same except for the following points:

* [anonymous\_method\_expression](#_Grm00064)s permit the parameter list to be omitted entirely, yielding convertibility to [delegate type](#_Trm00107)s of any list of [value](#_Trm00209) [parameters](#_Trm00059).
* [lambda\_expression](#_Grm00064)s permit parameter [types](#_Trm00011) to be omitted and inferred whereas [anonymous\_method\_expression](#_Grm00064)s require parameter [types](#_Trm00011) to be [explicit](#_Trm00198)ly stated.
* The body of a [lambda\_expression](#_Grm00064) can be an expression or a statement [block](#_Trm00038) whereas the body of an [anonymous\_method\_expression](#_Grm00064) must be a statement [block](#_Trm00038).
* Only [lambda\_expression](#_Grm00064)s have [conversion](#_Trm00196)s to compatible [expression tree types](#_Trm00184) ([§4.6](#_Toc00119)).

### Anonymous function [signature](#_Trm00061)s

The optional [anonymous\_function\_signature](#_Grm00064) of an [anonymous function](#_Trm00253) defines the names and optionally the [types](#_Trm00011) of the formal [parameters](#_Trm00059) for the [anonymous function](#_Trm00253). The [scope](#_Trm00148) of the [parameters](#_Trm00059) of the [anonymous function](#_Trm00253) is the [anonymous\_function\_body](#_Grm00064). ([§3.7](#_Toc00082)) Together with the parameter list (if given) the anonymous-[method](#_Trm00056)-body constitutes a [declaration space](#_Trm00130) ([§3.3](#_Toc00067)). It is thus a compile-time error for the name of a parameter of the [anonymous function](#_Trm00253) to match the name of a [local variable](#_Trm00193), local constant or parameter whose [scope](#_Trm00148) includes the [anonymous\_method\_expression](#_Grm00064) or [lambda\_expression](#_Grm00064).

If an [anonymous function](#_Trm00253) has an [explicit\_anonymous\_function\_signature](#_Grm00064), then the set of compatible [delegate type](#_Trm00107)s and [expression tree types](#_Trm00184) is restricted to those that have the same parameter [types](#_Trm00011) and modifiers in the same order. In contrast to [method](#_Trm00056) group [conversion](#_Trm00196)s ([§6.6](#_Toc00203)), contra-variance of [anonymous function](#_Trm00253) parameter [types](#_Trm00011) is not supported. If an [anonymous function](#_Trm00253) does not have an [anonymous\_function\_signature](#_Grm00064), then the set of compatible [delegate type](#_Trm00107)s and [expression tree types](#_Trm00184) is restricted to those that have no out [parameters](#_Trm00059).

Note that an [anonymous\_function\_signature](#_Grm00064) cannot include [attributes](#_Trm00108) or a parameter [array](#_Trm00093). Nevertheless, an [anonymous\_function\_signature](#_Grm00064) may be compatible with a [delegate type](#_Trm00107) whose parameter list contains a parameter [array](#_Trm00093).

Note also that [conversion](#_Trm00196) to an expression tree type, even if compatible, may still fail at compile-time ([§4.6](#_Toc00119)).

### Anonymous function bodies

The body ([expression](#_Grm00067) or [block](#_Grm00071)) of an [anonymous function](#_Trm00253) is subject to the following rules:

* If the [anonymous function](#_Trm00253) includes a [signature](#_Trm00061), the [parameters](#_Trm00059) specified in the [signature](#_Trm00061) are available in the body. If the [anonymous function](#_Trm00253) has no [signature](#_Trm00061) it can be converted to a [delegate type](#_Trm00107) or expression type having [parameters](#_Trm00059) ([§6.5](#_Toc00199)), but the [parameters](#_Trm00059) cannot be accessed in the body.
* Except for ref or out [parameters](#_Trm00059) specified in the [signature](#_Trm00061) (if any) of the nearest enclosing [anonymous function](#_Trm00253), 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](#_Trm00198) (as in this.x) or [implicit](#_Trm00197) (as in x where x is an [instance](#_Trm00172) 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](#_Trm00031) ([§7.15.5](#_Toc00326)) of the [anonymous function](#_Trm00253). Access of an outer variable will reference the [instance](#_Trm00172) of the variable that is active at the time the [lambda\_expression](#_Grm00064) or [anonymous\_method\_expression](#_Grm00064) is evaluated ([§7.15.6](#_Toc00329)).
* 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](#_Trm00253).
* A return statement in the body returns control from an invocation of the nearest enclosing [anonymous function](#_Trm00253), not from the enclosing function member. An expression specified in a return statement must be [implicit](#_Trm00197)ly convertible to the [return type](#_Trm00060) of the [delegate type](#_Trm00107) or expression tree type to which the nearest enclosing [lambda\_expression](#_Grm00064) or [anonymous\_method\_expression](#_Grm00064) is converted ([§6.5](#_Toc00199)).

It is [explicit](#_Trm00198)ly unspecified whether there is any way to execute the [block](#_Trm00038) of an [anonymous function](#_Trm00253) other than through evaluation and invocation of the [lambda\_expression](#_Grm00064) or [anonymous\_method\_expression](#_Grm00064). In particular, the compiler may choose to implement an [anonymous function](#_Trm00253) by synthesizing one or more named [method](#_Trm00056)s or [types](#_Trm00011). The names of any such synthesized [elements](#_Trm00094) must be of a form reserved for compiler use.

### Overload resolution and [anonymous function](#_Trm00253)s

Anonymous functions in an argument list participate in [type inference](#_Trm00231) and [overload resolution](#_Trm00078). Please refer to [§7.5.2](#_Toc00227) and [§7.5.3](#_Toc00242) for the exact rules.

The following example illustrates the effect of [anonymous function](#_Trm00253)s on [overload resolution](#_Trm00078).

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 [method](#_Trm00056)s. Each takes a selector argument, which extracts the [value](#_Trm00209) to sum over from a list item. The extracted [value](#_Trm00209) can be either an int or a double and the resulting sum is likewise either an int or a double.

The Sum [method](#_Trm00056)s 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 [method](#_Trm00056)s are applicable because the [anonymous function](#_Trm00253) d => d. UnitCount is compatible with both Func<Detail,int> and Func<Detail,double>. However, [overload resolution](#_Trm00078) picks the first Sum [method](#_Trm00056) because the [conversion](#_Trm00196) to Func<Detail,int> is better than the [conversion](#_Trm00196) to Func<Detail,double>.

In the second invocation of orderDetails.Sum, only the second Sum [method](#_Trm00056) is applicable because the [anonymous function](#_Trm00253) d => d.UnitPrice \* d.UnitCount produces a [value](#_Trm00209) of type double. Thus, [overload resolution](#_Trm00078) picks the second Sum [method](#_Trm00056) for that invocation.

### Anonymous functions and dynamic [binding](#_Trm00210)

An [anonymous function](#_Trm00253) cannot be a receiver, argument or operand of a dynamically bound operation.

### Outer [variables](#_Trm00031)

Any [local variable](#_Trm00193), [value](#_Trm00209) parameter, or parameter [array](#_Trm00093) whose [scope](#_Trm00148) includes the [lambda\_expression](#_Grm00064) or [anonymous\_method\_expression](#_Grm00064) is called an ***outer variable*** of the [anonymous function](#_Trm00253). In an [instance](#_Trm00172) function member of a class, the this [value](#_Trm00209) is considered a [value](#_Trm00209) parameter and is an [outer variable](#_Trm00254) of any [anonymous function](#_Trm00253) contained within the function member.

#### Captured [outer variable](#_Trm00254)s

When an [outer variable](#_Trm00254) is referenced by an [anonymous function](#_Trm00253), the [outer variable](#_Trm00254) is said to have been ***captured*** by the [anonymous function](#_Trm00253). Ordinarily, the lifetime of a [local variable](#_Trm00193) is limited to execution of the [block](#_Trm00038) or statement with which it is associated ([§5.1.7](#_Toc00130)). However, the lifetime of a [captured](#_Trm00255) [outer variable](#_Trm00254) is extended at least until the delegate or expression tree created from the [anonymous function](#_Trm00253) becomes [eligible](#_Trm00242) 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](#_Trm00193) x is [captured](#_Trm00255) by the [anonymous function](#_Trm00253), and the lifetime of x is extended at least until the delegate returned from F becomes [eligible](#_Trm00242) for garbage collection (which doesn't happen until the very end of the [program](#_Trm00109)). Since each invocation of the [anonymous function](#_Trm00253) operates on the same [instance](#_Trm00172) of x, the output of the example is:

1  
2  
3

When a [local variable](#_Trm00193) or a [value](#_Trm00209) parameter is [captured](#_Trm00255) by an [anonymous function](#_Trm00253), the [local variable](#_Trm00193) or parameter is no longer considered to be a fixed variable ([§18.3](#_Toc00593)), but is instead considered to be a moveable variable. Thus any unsafe code that takes the address of a [captured](#_Trm00255) [outer variable](#_Trm00254) must first use the fixed statement to fix the variable.

Note that unlike an un[captured](#_Trm00255) variable, a [captured](#_Trm00255) [local variable](#_Trm00193) can be simultaneously exposed to multiple threads of execution.

#### Instantiation of [local variable](#_Trm00193)s

A [local variable](#_Trm00193) is considered to be ***instantiated*** when execution enters the [scope](#_Trm00148) of the variable. For example, when the following [method](#_Trm00056) is invoked, the [local variable](#_Trm00193) x is [instantiated](#_Trm00256) 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](#_Trm00255), there is no way to observe exactly how often a [local variable](#_Trm00193) is [instantiated](#_Trm00256)—because the lifetimes of the instantiations are disjoint, it is possible for each instantation to simply use the same storage location. However, when an [anonymous function](#_Trm00253) captures a [local variable](#_Trm00193), 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](#_Trm00172) of the iteration variable is [captured](#_Trm00255), which produces the output:

3  
3  
3

It is possible for [anonymous function](#_Trm00253) delegates to share some [captured](#_Trm00255) [variables](#_Trm00031) yet have separate [instance](#_Trm00172)s 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](#_Trm00172) of x but separate [instance](#_Trm00172)s of y, and the output is:

1 1  
2 1  
3 1

Separate [anonymous function](#_Trm00253)s can capture the same [instance](#_Trm00172) of an [outer variable](#_Trm00254). 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 function](#_Trm00253)s capture the same [instance](#_Trm00172) of the [local variable](#_Trm00193) x, and they can thus "communicate" through that variable. The output of the example is:

5  
10

### Evaluation of [anonymous function](#_Trm00253) expressions

An [anonymous function](#_Trm00253) F must always be converted to a [delegate type](#_Trm00107) D or an expression tree type E, either directly or through the execution of a delegate creation expression new D(F). This [conversion](#_Trm00196) determines the result of the [anonymous function](#_Trm00253), as described in [§6.5](#_Toc00199).

## 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](#_Trm00094) of a ***sequence***. Each let clause introduces a [range variable](#_Trm00258) representing a [value](#_Trm00209) computed by means of previous [range variable](#_Trm00258)s. Each where clause is a filter that excludes items from the result. Each join clause compares specified keys of the source [sequence](#_Trm00259) with keys of another [sequence](#_Trm00259), 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 variable](#_Trm00258)s. 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](#_Trm00257) contain a number of "contextual [keyword](#_Trm00117)s", 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 [keyword](#_Trm00117)s or simple names, these identifiers are considered [keyword](#_Trm00117)s when occurring anywhere within a query expression.

For this purpose, a query expression is any expression that starts with "from dentifier" followed by any token except ";", "=" or ",".

In order to use these words as identifiers within a query expression, they can be prefixed with "@" ([§2.4.2](#_Toc00044)).

### Query expression translation

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of [method](#_Trm00056)s that adhere to the *query expression pattern* ([§7.16.3](#_Toc00340)). Specifically, query expressions are translated into invocations of [method](#_Trm00056)s named Where, Select, SelectMany, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, GroupBy, and Cast.These [method](#_Trm00056)s are expected to have particular [signature](#_Trm00061)s and result [types](#_Trm00011), as described in [§7.16.3](#_Toc00340). These [method](#_Trm00056)s can be [instance](#_Trm00172) [method](#_Trm00056)s of the [object](#_Trm00173) being queried or extension [method](#_Trm00056)s that are external to the [object](#_Trm00173), and they implement the actual execution of the query.

The translation from query expressions to [method](#_Trm00056) invocations is a syntactic mapping that occurs before any type [binding](#_Trm00210) or [overload resolution](#_Trm00078) 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](#_Trm00056) invocations are processed as regular [method](#_Trm00056) invocations, and this may in turn uncover errors, for example if the [method](#_Trm00056)s do not exist, if [arguments](#_Trm00062) have wrong [types](#_Trm00011), or if the [method](#_Trm00056)s are generic and [type inference](#_Trm00231) 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](#_Trm00124): 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 variable](#_Trm00258)s 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 variable](#_Trm00258)s with transparent identifiers denoted by \*. The special properties of transparent identifiers are discussed further in [§7.16.2.7](#_Toc00339).

#### 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](#_Trm00258) [types](#_Trm00011)

A from clause that [explicit](#_Trm00198)ly specifies a [range variable](#_Trm00258) type

from T x in e

is translated into

from x in ( e ) . Cast < T > ( )

A join clause that [explicit](#_Trm00198)ly specifies a [range variable](#_Trm00258) 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](#_Trm00198) [range variable](#_Trm00258) [types](#_Trm00011).

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](#_Trm00258) [types](#_Trm00011) are useful for querying collections that implement the non-generic IEnumerable [interface](#_Trm00102), but not the generic IEnumerable<T> [interface](#_Trm00102). 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](#_Trm00094) 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](#_Trm00173) 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 [explicit](#_Trm00198)ly calling Select on the source. It is then up to the implementers of Select and other query [operator](#_Trm00090)s to ensure that these [method](#_Trm00056)s never return the source [object](#_Trm00173) 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 in[visible](#_Trm00152) and in[accessible](#_Trm00138).

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 in[visible](#_Trm00152) and in[accessible](#_Trm00138).

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 in[visible](#_Trm00152) and in[accessible](#_Trm00138).

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 variable](#_Trm00258)s 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 function](#_Trm00253)s and anonymous [object](#_Trm00173) initializers. In those contexts, [transparent identifiers](#_Trm00260) have the following behavior:

* When a transparent identifier occurs as a parameter in an [anonymous function](#_Trm00253), the [members](#_Trm00012) of the associated anonymous type are automatically in [scope](#_Trm00148) in the body of the [anonymous function](#_Trm00253).
* When a member with a transparent identifier is in [scope](#_Trm00148), the [members](#_Trm00012) of that member are in [scope](#_Trm00148) as well.
* When a transparent identifier occurs as a member declarator in an anonymous [object](#_Trm00173) initializer, it introduces a member with a transparent identifier.
* In the translation steps described above, [transparent identifiers](#_Trm00260) are always introduced together with anonymous [types](#_Trm00011), with the intent of capturing multiple [range variable](#_Trm00258)s as [members](#_Trm00012) of a single [object](#_Trm00173). An implementation of C# is permitted to use a different mechanism than anonymous [types](#_Trm00011) to group together multiple [range variable](#_Trm00258)s. The following translation examples assume that anonymous [types](#_Trm00011) are used, and show how [transparent identifiers](#_Trm00260) 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](#_Trm00260) 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 in[visible](#_Trm00152) and in[accessible](#_Trm00138).

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 in[visible](#_Trm00152) and in[accessible](#_Trm00138).

### The query expression pattern

The ***Query expression pattern*** establishes a pattern of [method](#_Trm00056)s that [types](#_Trm00011) can implement to support query expressions. Because query expressions are translated to [method](#_Trm00056) invocations by means of a syntactic mapping, [types](#_Trm00011) have considerable flexibility in how they implement the query expression pattern. For example, the [method](#_Trm00056)s of the pattern can be implemented as [instance](#_Trm00172) [method](#_Trm00056)s or as extension [method](#_Trm00056)s because the two have the same invocation syntax, and the [method](#_Trm00056)s can request delegates or expression trees because [anonymous function](#_Trm00253)s 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](#_Trm00011), but it is possible to implement the pattern for non-[generic types](#_Trm00158) 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 [method](#_Trm00056)s above use the generic [delegate type](#_Trm00107)s Func<T1,R> and Func<T1,T2,R>, but they could equally well have used other delegate or [expression tree types](#_Trm00184) with the same relationships in parameter and result [types](#_Trm00011).

Notice the recommended relationship between C<T> and O<T> which ensures that the ThenBy and ThenByDescending [method](#_Trm00056)s are available only on the result of an OrderBy or OrderByDescending. Also notice the recommended shape of the result of GroupBy -- a [sequence](#_Trm00259) of [sequence](#_Trm00259)s, where each inner [sequence](#_Trm00259) has an additional Key property.

The System.Linq namespace provides an implementation of the query [operator](#_Trm00090) pattern for any type that implements the System.Collections.Generic.IEnumerable<T> [interface](#_Trm00102).

## Assignment [operator](#_Trm00090)s

The assignment [operator](#_Trm00090)s assign a new [value](#_Trm00209) to a variable, a property, an [event](#_Trm00088), or an [indexer](#_Trm00087) 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](#_Trm00087) access, or an [event](#_Trm00088) access.

The = [operator](#_Trm00090) is called the ***simple assignment operator***. It assigns the [value](#_Trm00209) of the right operand to the variable, property, or [indexer](#_Trm00087) element given by the left operand. The left operand of the simple assignment [operator](#_Trm00090) may not be an [event](#_Trm00088) access (except as described in [§10.8.1](#_Toc00464)). The simple assignment [operator](#_Trm00090) is described in [§7.17.1](#_Toc00342).

The assignment [operator](#_Trm00090)s other than the = [operator](#_Trm00090) are called the ***compound assignment operators***. These [operator](#_Trm00090)s perform the indicated operation on the two [operands](#_Trm00033), and then assign the resulting [value](#_Trm00209) to the variable, property, or [indexer](#_Trm00087) element given by the left operand. The [compound assignment operators](#_Trm00263) are described in [§7.17.2](#_Toc00343).

The += and -= [operator](#_Trm00090)s with an [event](#_Trm00088) access expression as the left operand are called the *event assignment operators*. No other assignment [operator](#_Trm00090) is valid with an [event](#_Trm00088) access as the left operand. The [event](#_Trm00088) assignment [operator](#_Trm00090)s are described in [§7.17.3](#_Toc00344).

The assignment [operator](#_Trm00090)s are [right-associative](#_Trm00222), 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](#_Trm00090) is called the simple assignment [operator](#_Trm00090).

If the left operand of a simple assignment is of the form E.P or E[Ei] where E has the [compile-time type](#_Trm00074) dynamic, then the assignment is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the assignment expression is dynamic, and the resolution described below will take place at run-time based on the [run-time type](#_Trm00073) of E.

In a simple assignment, the right operand must be an expression that is [implicit](#_Trm00197)ly convertible to the type of the left operand. The operation assigns the [value](#_Trm00209) of the right operand to the variable, property, or [indexer](#_Trm00087) element given by the left operand.

The result of a simple assignment expression is the [value](#_Trm00209) assigned to the left operand. The result has the same type as the left operand and is always classified as a [value](#_Trm00209).

If the left operand is a property or [indexer](#_Trm00087) access, the property or [indexer](#_Trm00087) must have a set accessor. If this is not the case, a [binding](#_Trm00210)-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](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)).
  + If the variable given by x is an [array](#_Trm00093) element of a [reference\_type](#_Grm00030), a run-time check is performed to ensure that the [value](#_Trm00209) computed for y is compatible with the [array](#_Trm00093) [instance](#_Trm00172) of which x is an element. The check succeeds if y is null, or if an [implicit](#_Trm00197) reference [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) exists from the actual type of the [instance](#_Trm00172) referenced by y to the actual [element type](#_Trm00095) of the [array](#_Trm00093) [instance](#_Trm00172) containing x. Otherwise, a System.ArrayTypeMismatchException is thrown.
  + The [value](#_Trm00209) resulting from the evaluation and [conversion](#_Trm00196) of y is stored into the location given by the evaluation of x.
* If x is classified as a property or [indexer](#_Trm00087) access:
  + The [instance](#_Trm00172) expression (if x is not static) and the argument list (if x is an [indexer](#_Trm00087) 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](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)).
  + The set accessor of x is invoked with the [value](#_Trm00209) computed for y as its value argument.

The [array](#_Trm00093) co-variance rules ([§12.5](#_Toc00525)) permit a [value](#_Trm00209) of an [array](#_Trm00093) type A[] to be a reference to an [instance](#_Trm00172) of an [array](#_Trm00093) type B[], provided an [implicit](#_Trm00197) reference [conversion](#_Trm00196) exists from B to A. Because of these rules, assignment to an [array](#_Trm00093) element of a [reference\_type](#_Grm00030) requires a run-time check to ensure that the [value](#_Trm00209) being assigned is compatible with the [array](#_Trm00093) [instance](#_Trm00172). 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](#_Trm00172) of ArrayList cannot be stored in an element of a string[].

When a property or [indexer](#_Trm00087) declared in a [struct\_type](#_Grm00029) is the target of an assignment, the [instance](#_Trm00172) expression associated with the property or [indexer](#_Trm00087) access must be classified as a variable. If the [instance](#_Trm00172) expression is classified as a [value](#_Trm00209), a [binding](#_Trm00210)-time error occurs. Because of [§7.6.4](#_Toc00257), 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](#_Trm00031). 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](#_Trm00031).

### 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](#_Trm00074) dynamic, then the assignment is dynamically bound ([§7.2.2](#_Toc00209)). In this case the [compile-time type](#_Trm00074) of the assignment expression is dynamic, and the resolution described below will take place at run-time based on the [run-time type](#_Trm00073) of E.

An operation of the form x op= y is processed by applying [binary operator overload resolution](#_Trm00226) ([§7.3.4](#_Toc00215)) as if the operation was written x op y. Then,

* If the [return type](#_Trm00060) of the selected [operator](#_Trm00090) is [implicit](#_Trm00197)ly 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](#_Trm00090) is a pre[defined](#_Trm00121) [operator](#_Trm00090), if the [return type](#_Trm00060) of the selected [operator](#_Trm00090) is [explicit](#_Trm00198)ly convertible to the type of x, and if y is [implicit](#_Trm00197)ly convertible to the type of x or the [operator](#_Trm00090) is a shift [operator](#_Trm00090), 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](#_Trm00210)-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](#_Trm00056) returning int[], and B and C are [method](#_Trm00056)s returning int, the [method](#_Trm00056)s are invoked only once, in the order A, B, C.

When the left operand of a compound assignment is a property access or [indexer](#_Trm00087) access, the property or [indexer](#_Trm00087) must have both a get accessor and a set accessor. If this is not the case, a [binding](#_Trm00210)-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 pre[defined](#_Trm00121) [operator](#_Trm00090)s can be used as compound [operator](#_Trm00090)s when the left operand is of type sbyte, byte, short, ushort, or char. Even when both [arguments](#_Trm00062) are of one of those [types](#_Trm00011), the pre[defined](#_Trm00121) [operator](#_Trm00090)s produce a result of type int, as described in [§7.3.6.2](#_Toc00219). Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for pre[defined](#_Trm00121) [operator](#_Trm00090)s 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](#_Trm00090) +(int?,int?) is used.

### Event assignment

If the left operand of a += or -= [operator](#_Trm00090) is classified as an [event](#_Trm00088) access, then the expression is evaluated as follows:

* The [instance](#_Trm00172) expression, if any, of the [event](#_Trm00088) access is evaluated.
* The right operand of the += or -= [operator](#_Trm00090) is evaluated, and, if required, converted to the type of the left operand through an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)).
* An [event](#_Trm00088) accessor of the [event](#_Trm00088) is invoked, with argument list consisting of the right operand, after evaluation and, if necessary, [conversion](#_Trm00196). If the [operator](#_Trm00090) was +=, the add accessor is invoked; if the [operator](#_Trm00090) was -=, the remove accessor is invoked.

An [event](#_Trm00088) assignment expression does not yield a [value](#_Trm00209). Thus, an [event](#_Trm00088) assignment expression is valid only in the context of a [statement\_expression](#_Grm00078) ([§8.6](#_Toc00357)).

## Expression

An [expression](#_Grm00067) is either a [non\_assignment\_expression](#_Grm00067) or an [assignment](#_Grm00066).

expression:  
 | non\_assignment\_expression  
 | assignment  
 ;  
  
non\_assignment\_expression:  
 | conditional\_expression  
 | lambda\_expression  
 | query\_expression  
 ;

## Constant expressions

A [constant\_expression](#_Grm00068) is an expression that can be fully evaluated at compile-time.

constant\_expression:  
 | expression  
 ;

A constant expression must be the null [literal](#_Trm00118) or a [value](#_Trm00209) with one of the following [types](#_Trm00011): sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, object, string, or any enumeration type. Only the following con[structs](#_Trm00092) are permitted in constant expressions:

* Literals (including the null [literal](#_Trm00118)).
* References to const [members](#_Trm00012) of class and [struct types](#_Trm00022).
* References to [members](#_Trm00012) of enumeration [types](#_Trm00011).
* References to const [parameters](#_Trm00059) or [local variable](#_Trm00193)s
* Parenthesized sub-expressions, which are themselves constant expressions.
* Cast expressions, provided the [target type](#_Trm00203) is one of the [types](#_Trm00011) listed above.
* checked and unchecked expressions
* Default [value](#_Trm00209) expressions
* The pre[defined](#_Trm00121) +, -, !, and ~ unary [operator](#_Trm00090)s.
* The pre[defined](#_Trm00121) +, -, \*, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <=, and >= binary [operator](#_Trm00090)s, provided each operand is of a type listed above.
* The ?: conditional [operator](#_Trm00090).

The following [conversion](#_Trm00196)s are permitted in constant expressions:

* Identity [conversion](#_Trm00196)s
* Numeric [conversion](#_Trm00196)s
* Enumeration [conversion](#_Trm00196)s
* Constant expression [conversion](#_Trm00196)s
* Implicit and [explicit](#_Trm00198) reference [conversion](#_Trm00196)s, provided that the source of the [conversion](#_Trm00196)s is a constant expression that evaluates to the [null value](#_Trm00165).

Other [conversion](#_Trm00196)s including [boxing](#_Trm00029), un[boxing](#_Trm00029) and [implicit](#_Trm00197) reference [conversion](#_Trm00196)s of non-[null value](#_Trm00165)s are not permitted in constant expressions. For example:

class C {  
 const object i = 5; // error: boxing conversion not permitted  
 const object str = "hello"; // error: implicit reference conversion  
}

the initialization of iis an error because a [boxing](#_Trm00029) [conversion](#_Trm00196) is required. The initialization of str is an error because an [implicit](#_Trm00197) reference [conversion](#_Trm00196) from a non-[null value](#_Trm00165) 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 con[structs](#_Trm00092).

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 [explicit](#_Trm00198)ly placed in an unchecked context, overflows that occur in integral-type arithmetic operations and [conversion](#_Trm00196)s during the compile-time evaluation of the expression always cause compile-time errors ([§7.19](#_Toc00346)).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

* Constant declarations ([§10.4](#_Toc00430)).
* Enumeration member declarations ([§14.3](#_Toc00554)).
* Default [arguments](#_Trm00062) of formal parameter lists ([§10.6.1](#_Toc00442))
* case labels of a switch statement ([§8.7.2](#_Toc00360)).
* goto case [statements](#_Trm00037) ([§8.9.3](#_Toc00369)).
* Dimension [length](#_Trm00096)s in an [array](#_Trm00093) creation expression ([§7.6.10.4](#_Toc00274)) that includes an initializer.
* Attributes ([§17](#_Toc00567)).

An [implicit](#_Trm00197) constant expression [conversion](#_Trm00196) ([§6.1.9](#_Toc00177)) permits a constant expression of type int to be converted to sbyte, byte, short, ushort, uint, or ulong, provided the [value](#_Trm00209) of the constant expression is within the range of the destination type.

## Boolean expressions

A [boolean\_expression](#_Grm00069) is an expression that yields a result of type bool; either directly or through [application](#_Trm00124) of operator true in certain contexts as specified in the following.

boolean\_expression:  
 | expression  
 ;

The controlling conditional expression of an [if\_statement](#_Grm00080) ([§8.7.1](#_Toc00359)), [while\_statement](#_Grm00083) ([§8.8.1](#_Toc00362)), [do\_statement](#_Grm00084) ([§8.8.2](#_Toc00363)), or [for\_statement](#_Grm00085) ([§8.8.3](#_Toc00364)) is a [boolean\_expression](#_Grm00069). The controlling conditional expression of the ?: [operator](#_Trm00090) ([§7.14](#_Toc00320)) follows the same rules as a [boolean\_expression](#_Grm00069), but for reasons of [operator](#_Trm00090) [precedence](#_Trm00035) is classified as a [conditional\_or\_expression](#_Grm00061).

A [boolean\_expression](#_Grm00069) E is required to be able to produce a [value](#_Trm00209) of type bool, as follows:

* If E is [implicit](#_Trm00197)ly convertible to bool then at runtime that [implicit](#_Trm00197) [conversion](#_Trm00196) is applied.
* Otherwise, unary [operator](#_Trm00090) [overload resolution](#_Trm00078) ([§7.3.3](#_Toc00214)) is used to find a unique best implementation of [operator](#_Trm00090) true on E, and that implementation is applied at runtime.
* If no such [operator](#_Trm00090) is found, a [binding](#_Trm00210)-time error occurs.

The DBBool struct type in [§11.4.2](#_Toc00517) provides an example of a type that implements operator true and operator false.

# Statements

C# provides a variety of [statements](#_Trm00037). Most of these [statements](#_Trm00037) will be familiar to developers who have [program](#_Trm00109)med 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](#_Grm00070) nonterminal is used for [statements](#_Trm00037) that appear within other [statements](#_Trm00037). The use of [embedded\_statement](#_Grm00070) rather than [statement](#_Grm00070) excludes the use of declaration [statements](#_Trm00037) and labeled [statements](#_Trm00037) 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](#_Grm00070) rather than a [statement](#_Grm00070) 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](#_Trm00038), the example is valid.

## End points and reachability

Every statement has an ***end point***. In intuitive terms, the [end point](#_Trm00264) of a statement is the location that immediately follows the statement. The execution rules for composite [statements](#_Trm00037) ([statements](#_Trm00037) that contain embedded [statements](#_Trm00037)) specify the action that is taken when control reaches the [end point](#_Trm00264) of an embedded statement. For example, when control reaches the [end point](#_Trm00264) of a statement in a [block](#_Trm00038), control is transferred to the next statement in the [block](#_Trm00038).

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 un[reachable](#_Trm00265) because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is un[reachable](#_Trm00265). It is specifically not an error for a statement to be un[reachable](#_Trm00265).

To determine whether a particular statement or [end point](#_Trm00264) is [reachable](#_Trm00265), the compiler performs flow analysis according to the reachability rules [defined](#_Trm00121) for each statement. The flow analysis takes into account the [value](#_Trm00209)s of constant expressions ([§7.19](#_Toc00346)) that control the behavior of [statements](#_Trm00037), but the possible [value](#_Trm00209)s 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](#_Trm00209) 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](#_Trm00033) of the == [operator](#_Trm00090) are constants. As the constant expression is evaluated at compile-time, producing the [value](#_Trm00209) false, the Console.WriteLine invocation is considered un[reachable](#_Trm00265). However, if i is changed to be a [local variable](#_Trm00193)

void F() {  
 int i = 1;  
 if (i == 2) Console.WriteLine("reachable");  
}

the Console.WriteLine invocation is considered [reachable](#_Trm00265), even though, in reality, it will never be executed.

The [block](#_Grm00071) of a function member is always considered [reachable](#_Trm00265). By successively evaluating the reachability rules of each statement in a [block](#_Trm00038), 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](#_Trm00265) because the [block](#_Trm00038) of the F [method](#_Trm00056) is [reachable](#_Trm00265).
* The [end point](#_Trm00264) of the first Console.WriteLine expression statement is [reachable](#_Trm00265) because that statement is [reachable](#_Trm00265).
* The if statement is [reachable](#_Trm00265) because the [end point](#_Trm00264) of the first Console.WriteLine expression statement is [reachable](#_Trm00265).
* The second Console.WriteLine expression statement is [reachable](#_Trm00265) because the boolean expression of the if statement does not have the constant [value](#_Trm00209) false.

There are two situations in which it is a compile-time error for the [end point](#_Trm00264) of a statement to be [reachable](#_Trm00265):

* 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](#_Trm00264) of the statement list of a switch section to be [reachable](#_Trm00265). 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](#_Trm00264) of the [block](#_Trm00038) of a function member that computes a [value](#_Trm00209) to be [reachable](#_Trm00265). If this error occurs, it typically is an indication that a return statement is missing.

## Blocks

A [block](#_Grm00071) permits multiple [statements](#_Trm00037) to be written in contexts where a single statement is allowed.

block:  
 | '{' statement\_list? '}'  
 ;

A [block](#_Grm00071) consists of an optional [statement\_list](#_Grm00072) ([§8.2.1](#_Toc00351)), enclosed in braces. If the statement list is omitted, the [block](#_Trm00038) is said to be empty.

A [block](#_Trm00038) may contain declaration [statements](#_Trm00037) ([§8.5](#_Toc00354)). The [scope](#_Trm00148) of a [local variable](#_Trm00193) or constant declared in a [block](#_Trm00038) is the [block](#_Trm00038).

Within a [block](#_Trm00038), the meaning of a name used in an expression context must always be the same ([§7.6.2.1](#_Toc00255)).

A [block](#_Trm00038) is executed as follows:

* If the [block](#_Trm00038) is empty, control is transferred to the [end point](#_Trm00264) of the [block](#_Trm00038).
* If the [block](#_Trm00038) is not empty, control is transferred to the statement list. When and if control reaches the [end point](#_Trm00264) of the statement list, control is transferred to the [end point](#_Trm00264) of the [block](#_Trm00038).

The statement list of a [block](#_Trm00038) is [reachable](#_Trm00265) if the [block](#_Trm00038) itself is [reachable](#_Trm00265).

The [end point](#_Trm00264) of a [block](#_Trm00038) is [reachable](#_Trm00265) if the [block](#_Trm00038) is empty or if the [end point](#_Trm00264) of the statement list is [reachable](#_Trm00265).

A [block](#_Grm00071) that contains one or more yield [statements](#_Trm00037) ([§8.14](#_Toc00376)) is called an iterator [block](#_Trm00038). Iterator [block](#_Trm00038)s are used to implement [function members](#_Trm00079) as iterators ([§10.14](#_Toc00483)). Some additional restrictions apply to iterator [block](#_Trm00038)s:

* It is a compile-time error for a return statement to appear in an iterator [block](#_Trm00038) (but yield return [statements](#_Trm00037) are permitted).
* It is a compile-time error for an iterator [block](#_Trm00038) to contain an unsafe context ([§18.1](#_Toc00591)). An iterator [block](#_Trm00038) always defines a safe context, even when its declaration is [nested](#_Trm00143) in an unsafe context.

### Statement lists

A ***statement list*** consists of one or more [statements](#_Trm00037) written in [sequence](#_Trm00259). Statement lists occur in [block](#_Grm00071)s ([§8.2](#_Toc00350)) and in [switch\_block](#_Grm00081)s ([§8.7.2](#_Toc00360)).

statement\_list:  
 | statement+  
 ;

A [statement list](#_Trm00267) is executed by transferring control to the first statement. When and if control reaches the [end point](#_Trm00264) of a statement, control is transferred to the next statement. When and if control reaches the [end point](#_Trm00264) of the last statement, control is transferred to the [end point](#_Trm00264) of the [statement list](#_Trm00267).

A statement in a [statement list](#_Trm00267) is [reachable](#_Trm00265) if at least one of the following is true:

* The statement is the first statement and the [statement list](#_Trm00267) itself is [reachable](#_Trm00265).
* The [end point](#_Trm00264) of the preceding statement is [reachable](#_Trm00265).
* The statement is a labeled statement and the label is referenced by a [reachable](#_Trm00265) goto statement.

The [end point](#_Trm00264) of a [statement list](#_Trm00267) is [reachable](#_Trm00265) if the [end point](#_Trm00264) of the last statement in the list is [reachable](#_Trm00265).

## The empty statement

An [empty\_statement](#_Grm00073) 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](#_Trm00264) of the statement. Thus, the [end point](#_Trm00264) of an empty statement is [reachable](#_Trm00265) if the empty statement is [reachable](#_Trm00265).

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](#_Trm00038):

void F() {  
 ...  
 if (done) goto exit;  
 ...  
 exit: ;  
}

## Labeled [statements](#_Trm00037)

A [labeled\_statement](#_Grm00074) permits a statement to be prefixed by a label. Labeled [statements](#_Trm00037) are permitted in [block](#_Trm00038)s, but are not permitted as embedded [statements](#_Trm00037).

labeled\_statement:  
 | identifier ':' statement  
 ;

A labeled statement declares a label with the name given by the [identifier](#_Grm00007). The [scope](#_Trm00148) of a label is the whole [block](#_Trm00038) in which the label is declared, including any [nested](#_Trm00143) [block](#_Trm00038)s. It is a compile-time error for two labels with the same name to have overlapping [scope](#_Trm00148)s.

A label can be referenced from goto [statements](#_Trm00037) ([§8.9.3](#_Toc00369)) within the [scope](#_Trm00148) of the label. This means that goto [statements](#_Trm00037) can transfer control within [block](#_Trm00038)s and out of [block](#_Trm00038)s, but never into [block](#_Trm00038)s.

Labels have their own [declaration space](#_Trm00130) 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](#_Trm00265) if the label is referenced by a [reachable](#_Trm00265) goto statement. (Exception: If a goto statement is inside a try that includes a finally [block](#_Trm00038), and the labeled statement is outside the try, and the [end point](#_Trm00264) of the finally [block](#_Trm00038) is un[reachable](#_Trm00265), then the labeled statement is not [reachable](#_Trm00265) from that goto statement.)

## [Declaration statements](#_Trm00039)

A [declaration\_statement](#_Grm00075) declares a [local variable](#_Trm00193) or constant. [Declaration statements](#_Trm00039) are permitted in [block](#_Trm00038)s, but are not permitted as embedded [statements](#_Trm00037).

declaration\_statement:  
 | local\_variable\_declaration ';'  
 | local\_constant\_declaration ';'  
 ;

### Local variable declarations

A [local\_variable\_declaration](#_Grm00076) declares one or more [local variable](#_Trm00193)s.

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](#_Grm00076) of a [local\_variable\_declaration](#_Grm00076) either directly specifies the type of the [variables](#_Trm00031) 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](#_Grm00076)s, each of which introduces a new variable. A [local\_variable\_declarator](#_Grm00076) consists of an [identifier](#_Grm00007) that names the variable, optionally followed by an "=" token and a [local\_variable\_initializer](#_Grm00076) that gives the initial [value](#_Trm00209) of the variable.

In the context of a [local variable](#_Trm00193) declaration, the identifier var acts as a contextual [keyword](#_Trm00117) ([§2.4.3](#_Toc00045)).When the [local\_variable\_type](#_Grm00076) is specified as var and no type named var is in [scope](#_Trm00148), 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](#_Trm00193) declarations are subject to the following restrictions:

* The [local\_variable\_declaration](#_Grm00076) cannot include multiple [local\_variable\_declarator](#_Grm00076)s.
* The [local\_variable\_declarator](#_Grm00076) must include a [local\_variable\_initializer](#_Grm00076).
* The [local\_variable\_initializer](#_Grm00076) must be an [expression](#_Grm00067).
* The initializer [expression](#_Grm00067) must have a [compile-time type](#_Trm00074).
* The initializer [expression](#_Grm00067) cannot refer to the declared variable itself

The following are examples of incorrect [implicit](#_Trm00197)ly typed [local variable](#_Trm00193) 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](#_Trm00209) of a [local variable](#_Trm00193) is obtained in an expression using a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)), and the [value](#_Trm00209) of a [local variable](#_Trm00193) is modified using an [assignment](#_Grm00066) ([§7.17](#_Toc00341)). A [local variable](#_Trm00193) must be [definitely assigned](#_Trm00068) ([§5.3](#_Toc00132)) at each location where its [value](#_Trm00209) is obtained.

The [scope](#_Trm00148) of a [local variable](#_Trm00193) declared in a [local\_variable\_declaration](#_Grm00076) is the [block](#_Trm00038) in which the declaration occurs. It is an error to refer to a [local variable](#_Trm00193) in a textual position that precedes the [local\_variable\_declarator](#_Grm00076) of the [local variable](#_Trm00193). Within the [scope](#_Trm00148) of a [local variable](#_Trm00193), it is a compile-time error to declare another [local variable](#_Trm00193) or constant with the same name.

A [local variable](#_Trm00193) declaration that declares multiple [variables](#_Trm00031) is equivalent to multiple declarations of single [variables](#_Trm00031) with the same type. Furthermore, a variable initializer in a [local variable](#_Trm00193) 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 [implicit](#_Trm00197)ly typed [local variable](#_Trm00193) declaration, the type of the [local variable](#_Trm00193) 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 [implicit](#_Trm00197)ly typed [local variable](#_Trm00193) declarations above are precisely equivalent to the following [explicit](#_Trm00198)ly 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](#_Grm00077) 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](#_Grm00028) of a [local\_constant\_declaration](#_Grm00077) specifies the type of the constants introduced by the declaration. The type is followed by a list of [constant\_declarator](#_Grm00077)s, each of which introduces a new constant. A [constant\_declarator](#_Grm00077) consists of an [identifier](#_Grm00007) that names the constant, followed by an "=" token, followed by a [constant\_expression](#_Grm00068) ([§7.19](#_Toc00346)) that gives the [value](#_Trm00209) of the constant.

The [type](#_Grm00028) and [constant\_expression](#_Grm00068) of a local constant declaration must follow the same rules as those of a constant member declaration ([§10.4](#_Toc00430)).

The [value](#_Trm00209) of a local constant is obtained in an expression using a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)).

The [scope](#_Trm00148) of a local constant is the [block](#_Trm00038) in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its [constant\_declarator](#_Grm00077). Within the [scope](#_Trm00148) of a local constant, it is a compile-time error to declare another [local variable](#_Trm00193) 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](#_Trm00040)

An [expression\_statement](#_Grm00078) evaluates a given expression. The [value](#_Trm00209) computed by the expression, if any, is discarded.

expression\_statement:  
 | statement\_expression ';'  
 ;  
  
statement\_expression:  
 | invocation\_expression  
 | object\_creation\_expression  
 | assignment  
 | post\_increment\_expression  
 | post\_decrement\_expression  
 | pre\_increment\_expression  
 | pre\_decrement\_expression  
 | await\_expression  
 ;

Not all expressions are permitted as [statements](#_Trm00037). In particular, expressions such as x + y and x == 1 that merely compute a [value](#_Trm00209) (which will be discarded), are not permitted as [statements](#_Trm00037).

Execution of an [expression\_statement](#_Grm00078) evaluates the contained expression and then transfers control to the [end point](#_Trm00264) of the [expression\_statement](#_Grm00078). The [end point](#_Trm00264) of an [expression\_statement](#_Grm00078) is [reachable](#_Trm00265) if that [expression\_statement](#_Grm00078) is [reachable](#_Trm00265).

## [Selection statements](#_Trm00041)

[Selection statements](#_Trm00041) select one of a number of possible [statements](#_Trm00037) for execution based on the [value](#_Trm00209) of some expression.

selection\_statement:  
 | if\_statement  
 | switch\_statement  
 ;

### The if statement

The if statement selects a statement for execution based on the [value](#_Trm00209) 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](#_Grm00069) ([§7.20](#_Toc00347)) is evaluated.
* If the boolean expression yields true, control is transferred to the first embedded statement. When and if control reaches the [end point](#_Trm00264) of that statement, control is transferred to the [end point](#_Trm00264) 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](#_Trm00264) of that statement, control is transferred to the [end point](#_Trm00264) 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](#_Trm00264) of the if statement.

The first embedded statement of an if statement is [reachable](#_Trm00265) if the if statement is [reachable](#_Trm00265) and the boolean expression does not have the constant [value](#_Trm00209) false.

The second embedded statement of an if statement, if present, is [reachable](#_Trm00265) if the if statement is [reachable](#_Trm00265) and the boolean expression does not have the constant [value](#_Trm00209) true.

The [end point](#_Trm00264) of an if statement is [reachable](#_Trm00265) if the [end point](#_Trm00264) of at least one of its embedded [statements](#_Trm00037) is [reachable](#_Trm00265). In addition, the [end point](#_Trm00264) of an if statement with no else part is [reachable](#_Trm00265) if the if statement is [reachable](#_Trm00265) and the boolean expression does not have the constant [value](#_Trm00209) true.

### The switch statement

The switch statement selects for execution a [statement list](#_Trm00267) having an associated switch label that corresponds to the [value](#_Trm00209) 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](#_Grm00081) consists of the [keyword](#_Trm00117) switch, followed by a parenthesized expression (called the switch expression), followed by a [switch\_block](#_Grm00081). The [switch\_block](#_Grm00081) consists of zero or more [switch\_section](#_Grm00081)s, enclosed in braces. Each [switch\_section](#_Grm00081) consists of one or more [switch\_label](#_Grm00081)s followed by a [statement\_list](#_Grm00072) ([§8.2.1](#_Toc00351)).

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](#_Grm00029), or if it is the nullable type corresponding to one of these [types](#_Trm00011), then that is the [governing type](#_Trm00269) of the switch statement.
* Otherwise, exactly one user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.4](#_Toc00193)) must exist from the type of the switch expression to one of the following possible [governing type](#_Trm00269)s: sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or, a nullable type corresponding to one of those [types](#_Trm00011).
* Otherwise, if no such [implicit](#_Trm00197) [conversion](#_Trm00196) exists, or if more than one such [implicit](#_Trm00197) [conversion](#_Trm00196) exists, a compile-time error occurs.

The constant expression of each case label must denote a [value](#_Trm00209) that is [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the [governing type](#_Trm00269) 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](#_Trm00209).

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](#_Trm00269).
* If one of the constants specified in a case label in the same switch statement is equal to the [value](#_Trm00209) of the switch expression, control is transferred to the [statement list](#_Trm00267) following the matched case label.
* If none of the constants specified in case labels in the same switch statement is equal to the [value](#_Trm00209) of the switch expression, and if a default label is present, control is transferred to the [statement list](#_Trm00267) following the default label.
* If none of the constants specified in case labels in the same switch statement is equal to the [value](#_Trm00209) of the switch expression, and if no default label is present, control is transferred to the [end point](#_Trm00264) of the switch statement.

If the [end point](#_Trm00264) of the [statement list](#_Trm00267) of a switch section is [reachable](#_Trm00265), 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](#_Trm00265) [end point](#_Trm00264). 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](#_Trm00198) 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](#_Grm00081). 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](#_Grm00081).

The "no fall through" rule pr[event](#_Trm00088)s a common class of bugs that occur in C and C++ when break [statements](#_Trm00037) 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](#_Trm00267) of a switch section typically ends in a break, goto case, or goto default statement, but any construct that renders the [end point](#_Trm00264) of the [statement list](#_Trm00267) un[reachable](#_Trm00265) is permitted. For example, a while statement controlled by the boolean expression true is known to never reach its [end point](#_Trm00264). Likewise, a throw or return statement always transfers control elsewhere and never reaches its [end point](#_Trm00264). Thus, the following example is valid:

switch (i) {  
case 0:  
 while (true) F();  
case 1:  
 throw new ArgumentException();  
case 2:  
 return;  
}

The [governing type](#_Trm00269) 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 [operator](#_Trm00090)s ([§7.10.7](#_Toc00306)), 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](#_Trm00269) of a switch statement is string, the [value](#_Trm00209) null is permitted as a case label constant.

The [statement\_list](#_Grm00072)s of a [switch\_block](#_Grm00081) may contain declaration [statements](#_Trm00037) ([§8.5](#_Toc00354)). The [scope](#_Trm00148) of a [local variable](#_Trm00193) or constant declared in a switch [block](#_Trm00038) is the switch [block](#_Trm00038).

Within a switch [block](#_Trm00038), the meaning of a name used in an expression context must always be the same ([§7.6.2.1](#_Toc00255)).

The [statement list](#_Trm00267) of a given switch section is [reachable](#_Trm00265) if the switch statement is [reachable](#_Trm00265) and at least one of the following is true:

* The switch expression is a non-constant [value](#_Trm00209).
* The switch expression is a constant [value](#_Trm00209) that matches a case label in the switch section.
* The switch expression is a constant [value](#_Trm00209) 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](#_Trm00265) goto case or goto default statement.

The [end point](#_Trm00264) of a switch statement is [reachable](#_Trm00265) if at least one of the following is true:

* The switch statement contains a [reachable](#_Trm00265) break statement that exits the switch statement.
* The switch statement is [reachable](#_Trm00265), the switch expression is a non-constant [value](#_Trm00209), and no default label is present.
* The switch statement is [reachable](#_Trm00265), the switch expression is a constant [value](#_Trm00209) that doesn't match any case label, and no default label is present.

## [Iteration statements](#_Trm00042)

[Iteration statements](#_Trm00042) 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](#_Grm00069) ([§7.20](#_Toc00347)) is evaluated.
* If the boolean expression yields true, control is transferred to the embedded statement. When and if control reaches the [end point](#_Trm00264) 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](#_Trm00264) of the while statement.

Within the embedded statement of a while statement, a break statement ([§8.9.1](#_Toc00367)) may be used to transfer control to the [end point](#_Trm00264) of the while statement (thus ending iteration of the embedded statement), and a continue statement ([§8.9.2](#_Toc00368)) may be used to transfer control to the [end point](#_Trm00264) of the embedded statement (thus performing another iteration of the while statement).

The embedded statement of a while statement is [reachable](#_Trm00265) if the while statement is [reachable](#_Trm00265) and the boolean expression does not have the constant [value](#_Trm00209) false.

The [end point](#_Trm00264) of a while statement is [reachable](#_Trm00265) if at least one of the following is true:

* The while statement contains a [reachable](#_Trm00265) break statement that exits the while statement.
* The while statement is [reachable](#_Trm00265) and the boolean expression does not have the constant [value](#_Trm00209) 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](#_Trm00264) of the embedded statement (possibly from execution of a continue statement), the [boolean\_expression](#_Grm00069) ([§7.20](#_Toc00347)) 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](#_Trm00264) of the do statement.

Within the embedded statement of a do statement, a break statement ([§8.9.1](#_Toc00367)) may be used to transfer control to the [end point](#_Trm00264) of the do statement (thus ending iteration of the embedded statement), and a continue statement ([§8.9.2](#_Toc00368)) may be used to transfer control to the [end point](#_Trm00264) of the embedded statement.

The embedded statement of a do statement is [reachable](#_Trm00265) if the do statement is [reachable](#_Trm00265).

The [end point](#_Trm00264) of a do statement is [reachable](#_Trm00265) if at least one of the following is true:

* The do statement contains a [reachable](#_Trm00265) break statement that exits the do statement.
* The [end point](#_Trm00264) of the embedded statement is [reachable](#_Trm00265) and the boolean expression does not have the constant [value](#_Trm00209) true.

### The for statement

The for statement evaluates a [sequence](#_Trm00259) of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a [sequence](#_Trm00259) 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](#_Grm00085), if present, consists of either a [local\_variable\_declaration](#_Grm00076) ([§8.5.1](#_Toc00355)) or a list of [statement\_expression](#_Grm00078)s ([§8.6](#_Toc00357)) separated by commas. The [scope](#_Trm00148) of a [local variable](#_Trm00193) declared by a [for\_initializer](#_Grm00085) starts at the [local\_variable\_declarator](#_Grm00076) for the variable and extends to the end of the embedded statement. The [scope](#_Trm00148) includes the [for\_condition](#_Grm00085) and the [for\_iterator](#_Grm00085).

The [for\_condition](#_Grm00085), if present, must be a [boolean\_expression](#_Grm00069) ([§7.20](#_Toc00347)).

The [for\_iterator](#_Grm00085), if present, consists of a list of [statement\_expression](#_Grm00078)s ([§8.6](#_Toc00357)) separated by commas.

A for statement is executed as follows:

* If a [for\_initializer](#_Grm00085) 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](#_Grm00085) is present, it is evaluated.
* If the [for\_condition](#_Grm00085) is not present or if the evaluation yields true, control is transferred to the embedded statement. When and if control reaches the [end point](#_Trm00264) of the embedded statement (possibly from execution of a continue statement), the expressions of the [for\_iterator](#_Grm00085), if any, are evaluated in [sequence](#_Trm00259), and then another iteration is performed, starting with evaluation of the [for\_condition](#_Grm00085) in the step above.
* If the [for\_condition](#_Grm00085) is present and the evaluation yields false, control is transferred to the [end point](#_Trm00264) of the for statement.

Within the embedded statement of a for statement, a break statement ([§8.9.1](#_Toc00367)) may be used to transfer control to the [end point](#_Trm00264) of the for statement (thus ending iteration of the embedded statement), and a continue statement ([§8.9.2](#_Toc00368)) may be used to transfer control to the [end point](#_Trm00264) of the embedded statement (thus executing the [for\_iterator](#_Grm00085) and performing another iteration of the for statement, starting with the [for\_condition](#_Grm00085)).

The embedded statement of a for statement is [reachable](#_Trm00265) if one of the following is true:

* The for statement is [reachable](#_Trm00265) and no [for\_condition](#_Grm00085) is present.
* The for statement is [reachable](#_Trm00265) and a [for\_condition](#_Grm00085) is present and does not have the constant [value](#_Trm00209) false.

The [end point](#_Trm00264) of a for statement is [reachable](#_Trm00265) if at least one of the following is true:

* The for statement contains a [reachable](#_Trm00265) break statement that exits the for statement.
* The for statement is [reachable](#_Trm00265) and a [for\_condition](#_Grm00085) is present and does not have the constant [value](#_Trm00209) true.

### The foreach statement

The foreach statement enumerates the [elements](#_Trm00094) 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](#_Grm00028) and [identifier](#_Grm00007) of a foreach statement declare the ***iteration variable*** of the statement. If the var identifier is given as the [local\_variable\_type](#_Grm00076), and no type named var is in [scope](#_Trm00148), the [iteration variable](#_Trm00270) is said to be an ***implicitly typed iteration variable***, and its type is taken to be the [element type](#_Trm00095) of the foreach statement, as specified below. The [iteration variable](#_Trm00270) corresponds to a read-only [local variable](#_Trm00193) with a [scope](#_Trm00148) that extends over the embedded statement. During execution of a foreach statement, the [iteration variable](#_Trm00270) 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](#_Trm00270) (via assignment or the ++ and -- [operator](#_Trm00090)s) or pass the [iteration variable](#_Trm00270) as a ref or out parameter.

In the following, for brevity, IEnumerable, IEnumerator, IEnumerable<T> and IEnumerator<T> refer to the corresponding [types](#_Trm00011) in the [namespaces](#_Trm00010) 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](#_Grm00067) is an [array](#_Trm00093) type then there is an [implicit](#_Trm00197) reference [conversion](#_Trm00196) from X to the IEnumerable [interface](#_Trm00102) (since System.Array implements this [interface](#_Trm00102)). The ***collection type*** is the IEnumerable [interface](#_Trm00102), the ***enumerator type*** is the IEnumerator [interface](#_Trm00102) and the ***element type*** is the [element type](#_Trm00095) of the [array](#_Trm00093) type X.
* If the type X of [expression](#_Grm00067) is dynamic then there is an [implicit](#_Trm00197) [conversion](#_Trm00196) from [expression](#_Grm00067) to the IEnumerable [interface](#_Trm00102) ([§6.1.8](#_Toc00176)). The ***collection type*** is the IEnumerable [interface](#_Trm00102) and the ***enumerator type*** is the IEnumerator [interface](#_Trm00102). If the var identifier is given as the [local\_variable\_type](#_Grm00076) then the ***element type*** is dynamic, otherwise it is object.
* Otherwise, determine whether the type X has an appropriate GetEnumerator [method](#_Trm00056):
  + Perform member lookup on the type X with identifier GetEnumerator and no type [arguments](#_Trm00062). If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a [method](#_Trm00056) group, check for an enumerable [interface](#_Trm00102) as described below. It is recommended that a warning be issued if member lookup produces anything except a [method](#_Trm00056) group or no match.
  + Perform [overload resolution](#_Trm00078) using the resulting [method](#_Trm00056) group and an empty argument list. If [overload resolution](#_Trm00078) results in no applicable [method](#_Trm00056)s, results in an ambiguity, or results in a single best [method](#_Trm00056) but that [method](#_Trm00056) is either static or not public, check for an enumerable [interface](#_Trm00102) as described below. It is recommended that a warning be issued if [overload resolution](#_Trm00078) produces anything except an unambiguous public [instance](#_Trm00172) [method](#_Trm00056) or no applicable [method](#_Trm00056)s.
  + If the [return type](#_Trm00060) E of the GetEnumerator [method](#_Trm00056) is not a class, struct or [interface](#_Trm00102) 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](#_Trm00062). If the member lookup produces no match, the result is an error, or the result is anything except a public [instance](#_Trm00172) 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](#_Trm00062). If the member lookup produces no match, the result is an error, or the result is anything except a [method](#_Trm00056) group, an error is produced and no further steps are taken.
  + Overload resolution is performed on the [method](#_Trm00056) group with an empty argument list. If [overload resolution](#_Trm00078) results in no applicable [method](#_Trm00056)s, results in an ambiguity, or results in a single best [method](#_Trm00056) but that [method](#_Trm00056) is either static or not public, or its [return type](#_Trm00060) 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](#_Trm00102):
  + If among all the [types](#_Trm00011) Ti for which there is an [implicit](#_Trm00197) [conversion](#_Trm00196) 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](#_Trm00197) [conversion](#_Trm00196) from IEnumerable<T> to IEnumerable<Ti>, then the ***collection type*** is the [interface](#_Trm00102) IEnumerable<T>, the ***enumerator type*** is the [interface](#_Trm00102) 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](#_Trm00197) [conversion](#_Trm00196) from X to the System.Collections.IEnumerable [interface](#_Trm00102), then the ***collection type*** is this [interface](#_Trm00102), the ***enumerator type*** is the [interface](#_Trm00102) 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](#_Trm00272) C, [enumerator type](#_Trm00273) E and [element type](#_Trm00095) 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](#_Trm00152) to or [accessible](#_Trm00138) to the expression x or the embedded statement or any other source code of the [program](#_Trm00109). The variable v is read-only in the embedded statement. If there is not an [explicit](#_Trm00198) [conversion](#_Trm00196) ([§6.2](#_Toc00181)) from T (the [element type](#_Trm00095)) to V (the [local\_variable\_type](#_Grm00076) in the foreach statement), an error is produced and no further steps are taken. If x has the [value](#_Trm00209) 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](#_Trm00255) by any [anonymous function](#_Trm00253) occurring in the [embedded\_statement](#_Grm00070).

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](#_Trm00209) after the for loop would be the final [value](#_Trm00209), 13, which is what the invocation of f would print. Instead, because each iteration has its own variable v, the one [captured](#_Trm00255) by f in the first iteration will continue to hold the [value](#_Trm00209) 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](#_Trm00038) is constructed according to the following steps:

* If there is an [implicit](#_Trm00197) [conversion](#_Trm00196) from E to the System.IDisposable [interface](#_Trm00102), then
  + If E is a [non-nullable value type](#_Trm00169) 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](#_Trm00209) type, or a type parameter [instantiated](#_Trm00256) to a [value](#_Trm00209) type, then the cast of e to System.IDisposable will not cause [boxing](#_Trm00029) to occur.

* Otherwise, if E is a sealed type, the finally clause is expanded to an empty [block](#_Trm00038):

finally {  
}

* Otherwise, the finally clause is expanded to:

finally {  
 System.IDisposable d = e as System.IDisposable;  
 if (d != null) d.Dispose();  
}

The [local variable](#_Trm00193) d is not [visible](#_Trm00152) to or [accessible](#_Trm00138) to any user code. In particular, it does not conflict with any other variable whose [scope](#_Trm00148) includes the finally [block](#_Trm00038).

The order in which foreach traverses the [elements](#_Trm00094) of an [array](#_Trm00093), is as follows: For single-dimensional [array](#_Trm00093)s [elements](#_Trm00094) are traversed in increasing index order, starting with index 0 and ending with index Length - 1. For multi-dimensional [array](#_Trm00093)s, [elements](#_Trm00094) 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](#_Trm00209) in a two-dimensional [array](#_Trm00093), 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](#_Trm00095) of numbers.

## [Jump statements](#_Trm00043)

[Jump statements](#_Trm00043) 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](#_Trm00038), and the [target](#_Trm00290) of that jump statement is outside that [block](#_Trm00038), the jump statement is said to ***exit*** the [block](#_Trm00038). While a jump statement may transfer control out of a [block](#_Trm00038), it can never transfer control into a [block](#_Trm00038).

Execution of jump [statements](#_Trm00037) is complicated by the presence of intervening try [statements](#_Trm00037). In the absence of such try [statements](#_Trm00037), a jump statement unconditionally transfers control from the jump statement to its [target](#_Trm00290). In the presence of such intervening try [statements](#_Trm00037), execution is more complex. If the jump statement [exit](#_Trm00291)s one or more try [block](#_Trm00038)s with associated finally [block](#_Trm00038)s, control is initially transferred to the finally [block](#_Trm00038) of the innermost try statement. When and if control reaches the [end point](#_Trm00264) of a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038) of the next enclosing try statement. This process is repeated until the finally [block](#_Trm00038)s of all intervening try [statements](#_Trm00037) 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 [block](#_Trm00038)s associated with two try [statements](#_Trm00037) are executed before control is transferred to the [target](#_Trm00290) 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 [exit](#_Trm00291)s the nearest enclosing switch, while, do, for, or foreach statement.

break\_statement:  
 | 'break' ';'  
 ;

The [target](#_Trm00290) of a break statement is the [end point](#_Trm00264) 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](#_Trm00037) are [nested](#_Trm00143) within each other, a break statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement ([§8.9.3](#_Toc00369)) must be used.

A break statement cannot [exit](#_Trm00291) a finally [block](#_Trm00038) ([§8.10](#_Toc00372)). When a break statement occurs within a finally [block](#_Trm00038), the [target](#_Trm00290) of the break statement must be within the same finally [block](#_Trm00038); otherwise, a compile-time error occurs.

A break statement is executed as follows:

* If the break statement [exit](#_Trm00291)s one or more try [block](#_Trm00038)s with associated finally [block](#_Trm00038)s, control is initially transferred to the finally [block](#_Trm00038) of the innermost try statement. When and if control reaches the [end point](#_Trm00264) of a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038) of the next enclosing try statement. This process is repeated until the finally [block](#_Trm00038)s of all intervening try [statements](#_Trm00037) have been executed.
* Control is transferred to the [target](#_Trm00290) of the break statement.

Because a break statement unconditionally transfers control elsewhere, the [end point](#_Trm00264) of a break statement is never [reachable](#_Trm00265).

### 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](#_Trm00290) of a continue statement is the [end point](#_Trm00264) 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](#_Trm00037) are [nested](#_Trm00143) within each other, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement ([§8.9.3](#_Toc00369)) must be used.

A continue statement cannot [exit](#_Trm00291) a finally [block](#_Trm00038) ([§8.10](#_Toc00372)). When a continue statement occurs within a finally [block](#_Trm00038), the [target](#_Trm00290) of the continue statement must be within the same finally [block](#_Trm00038); otherwise a compile-time error occurs.

A continue statement is executed as follows:

* If the continue statement [exit](#_Trm00291)s one or more try [block](#_Trm00038)s with associated finally [block](#_Trm00038)s, control is initially transferred to the finally [block](#_Trm00038) of the innermost try statement. When and if control reaches the [end point](#_Trm00264) of a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038) of the next enclosing try statement. This process is repeated until the finally [block](#_Trm00038)s of all intervening try [statements](#_Trm00037) have been executed.
* Control is transferred to the [target](#_Trm00290) of the continue statement.

Because a continue statement unconditionally transfers control elsewhere, the [end point](#_Trm00264) of a continue statement is never [reachable](#_Trm00265).

### 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](#_Trm00290) of a goto [identifier](#_Grm00007) 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](#_Trm00148) of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control out of a [nested](#_Trm00143) [scope](#_Trm00148), but not into a [nested](#_Trm00143) [scope](#_Trm00148). 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](#_Trm00143) [scope](#_Trm00148).

The [target](#_Trm00290) of a goto case statement is the [statement list](#_Trm00267) in the immediately enclosing switch statement ([§8.7.2](#_Toc00360)), which contains a case label with the given constant [value](#_Trm00209). If the goto case statement is not enclosed by a switch statement, if the [constant\_expression](#_Grm00068) is not [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the [governing type](#_Trm00269) of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant [value](#_Trm00209), a compile-time error occurs.

The [target](#_Trm00290) of a goto default statement is the [statement list](#_Trm00267) in the immediately enclosing switch statement ([§8.7.2](#_Toc00360)), 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](#_Trm00291) a finally [block](#_Trm00038) ([§8.10](#_Toc00372)). When a goto statement occurs within a finally [block](#_Trm00038), the [target](#_Trm00290) of the goto statement must be within the same finally [block](#_Trm00038), or otherwise a compile-time error occurs.

A goto statement is executed as follows:

* If the goto statement [exit](#_Trm00291)s one or more try [block](#_Trm00038)s with associated finally [block](#_Trm00038)s, control is initially transferred to the finally [block](#_Trm00038) of the innermost try statement. When and if control reaches the [end point](#_Trm00264) of a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038) of the next enclosing try statement. This process is repeated until the finally [block](#_Trm00038)s of all intervening try [statements](#_Trm00037) have been executed.
* Control is transferred to the [target](#_Trm00290) of the goto statement.

Because a goto statement unconditionally transfers control elsewhere, the [end point](#_Trm00264) of a goto statement is never [reachable](#_Trm00265).

### 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](#_Trm00209), that is, a [method](#_Trm00056) with the result type ([§10.6.10](#_Toc00455)) void, the set accessor of a property or [indexer](#_Trm00087), the add and remove [accessors](#_Trm00083) of an [event](#_Trm00088), an [instance](#_Trm00172) constructor, a [static constructor](#_Trm00081), or a [destructor](#_Trm00091).

A return statement with an expression can only be used in a function member that computes a [value](#_Trm00209), that is, a [method](#_Trm00056) with a non-void result type, the get accessor of a property or [indexer](#_Trm00087), or a user-[defined](#_Trm00121) [operator](#_Trm00090). An [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) must exist from the type of the expression to the [return type](#_Trm00060) of the containing function member.

Return [statements](#_Trm00037) can also be used in the body of [anonymous function](#_Trm00253) expressions ([§7.15](#_Toc00321)), and participate in determining which [conversion](#_Trm00196)s exist for those functions.

It is a compile-time error for a return statement to appear in a finally [block](#_Trm00038) ([§8.10](#_Toc00372)).

A return statement is executed as follows:

* If the return statement specifies an expression, the expression is evaluated and the resulting [value](#_Trm00209) is converted to the [return type](#_Trm00060) of the containing function by an [implicit](#_Trm00197) [conversion](#_Trm00196). The result of the [conversion](#_Trm00196) becomes the result [value](#_Trm00209) produced by the function.
* If the return statement is enclosed by one or more try or catch [block](#_Trm00038)s with associated finally [block](#_Trm00038)s, control is initially transferred to the finally [block](#_Trm00038) of the innermost try statement. When and if control reaches the [end point](#_Trm00264) of a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038) of the next enclosing try statement. This process is repeated until the finally [block](#_Trm00038)s of all enclosing try [statements](#_Trm00037) 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](#_Trm00209), if any.
* If the containing function is an async function, control is returned to the current caller, and the result [value](#_Trm00209), if any, is recorded in the return [task](#_Trm00248) as described in ([§10.14.1](#_Toc00484)).

Because a return statement unconditionally transfers control elsewhere, the [end point](#_Trm00264) of a return statement is never [reachable](#_Trm00265).

### The throw statement

The throw statement throws an exception.

throw\_statement:  
 | 'throw' expression? ';'  
 ;

A throw statement with an expression throws the [value](#_Trm00209) produced by evaluating the expression. The expression must denote a [value](#_Trm00209) 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](#_Trm00038), in which case that statement re-throws the exception that is currently being handled by that catch [block](#_Trm00038).

Because a throw statement unconditionally transfers control elsewhere, the [end point](#_Trm00264) of a throw statement is never [reachable](#_Trm00265).

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](#_Trm00293) 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](#_Trm00038) of S encloses the [throw point](#_Trm00293) and if S has one or more catch clauses, the catch clauses are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause ([§8.10](#_Toc00372)) is considered a match for any exception type. If a matching catch clause is located, the [exception propagation](#_Trm00292) is completed by transferring control to the [block](#_Trm00038) of that catch clause.
  + Otherwise, if the try [block](#_Trm00038) or a catch [block](#_Trm00038) of S encloses the [throw point](#_Trm00293) and if S has a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038). If the finally [block](#_Trm00038) throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the [end point](#_Trm00264) of the finally [block](#_Trm00038), 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](#_Trm00293) corresponding to the statement from which the function member was invoked.
  + If the current function is async and [task](#_Trm00248)-returning, the exception is recorded in the return [task](#_Trm00248), which is put into a faulted or cancelled state as described in [§10.14.1](#_Toc00484).
  + If the current function is async and void-returning, the synchronization context of the current thread is notified as described in [§10.14.2](#_Toc00485).
* 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](#_Trm00121).

## The try statement

The try statement provides a mechanism for catching exceptions that occur during execution of a [block](#_Trm00038). Furthermore, the try statement provides the ability to specify a [block](#_Trm00038) of code that is always executed when control leaves the try statement.

try\_statement:  
 | 'try' block catch\_clauses  
 | 'try' block finally\_clause  
 | 'try' block catch\_clauses finally\_clause  
 ;  
  
catch\_clauses:  
 | specific\_catch\_clause+ general\_catch\_clause?  
 | specific\_catch\_clause\* general\_catch\_clause  
 ;  
  
specific\_catch\_clause:  
 | 'catch' '(' class\_type identifier? ')' block  
 ;  
  
general\_catch\_clause:  
 | 'catch' block  
 ;  
  
finally\_clause:  
 | 'finally' block  
 ;

There are three possible forms of try [statements](#_Trm00037):

* A try [block](#_Trm00038) followed by one or more catch [block](#_Trm00038)s.
* A try [block](#_Trm00038) followed by a finally [block](#_Trm00038).
* A try [block](#_Trm00038) followed by one or more catch [block](#_Trm00038)s followed by a finally [block](#_Trm00038).

When a catch clause specifies a [class\_type](#_Grm00030), the type must be System.Exception, a type that derives from System.Exception or a type parameter type that has System.Exception (or a subclass thereof) as its effective base class.

When a catch clause specifies both a [class\_type](#_Grm00030) and an [identifier](#_Grm00007), an ***exception variable*** of the given name and type is declared. The [exception variable](#_Trm00294) corresponds to a [local variable](#_Trm00193) with a [scope](#_Trm00148) that extends over the catch [block](#_Trm00038). During execution of the catch [block](#_Trm00038), the [exception variable](#_Trm00294) represents the exception currently being handled. For purposes of definite assignment checking, the [exception variable](#_Trm00294) is considered [definitely assigned](#_Trm00068) in its entire [scope](#_Trm00148).

Unless a catch clause includes an [exception variable](#_Trm00294) name, it is impossible to access the exception [object](#_Trm00173) in the catch [block](#_Trm00038).

A catch clause that specifies neither an exception type nor an [exception variable](#_Trm00294) name is called a general catch clause. A try statement can only have one general catch clause, and if one is present it must be the last catch clause.

Some [program](#_Trm00109)ming languages may support exceptions that are not representable as an [object](#_Trm00173) derived from System.Exception, although such exceptions could never be generated by C# code. A general catch clause may be used to catch such exceptions. Thus, a general catch clause is semantically different from one that specifies the type System.Exception, in that the former may also catch exceptions from other languages.

In order to locate a handler for an exception, catch clauses are examined in lexical order. A compile-time error occurs if a catch clause specifies a type that is the same as, or is derived from, a type that was specified in an earlier catch clause for the same try. Without this restriction, it would be possible to write un[reachable](#_Trm00265) catch clauses.

Within a catch [block](#_Trm00038), a throw statement ([§8.9.5](#_Toc00371)) with no expression can be used to re-throw the exception that was caught by the catch [block](#_Trm00038). Assignments to an [exception variable](#_Trm00294) 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](#_Trm00056) F catches an exception, writes some diagnostic information to the console, alters the [exception variable](#_Trm00294), 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](#_Trm00038) had thrown e instead of rethrowing the current exception, the output produced is would be as follows:

Exception in F: G  
Exception in Main: F

It is a compile-time error for a break, continue, or goto statement to transfer control out of a finally [block](#_Trm00038). When a break, continue, or goto statement occurs in a finally [block](#_Trm00038), the [target](#_Trm00290) of the statement must be within the same finally [block](#_Trm00038), or otherwise a compile-time error occurs.

It is a compile-time error for a return statement to occur in a finally [block](#_Trm00038).

A try statement is executed as follows:

* Control is transferred to the try [block](#_Trm00038).
* When and if control reaches the [end point](#_Trm00264) of the try [block](#_Trm00038):
  + If the try statement has a finally [block](#_Trm00038), the finally [block](#_Trm00038) is executed.
  + Control is transferred to the [end point](#_Trm00264) of the try statement.
* If an exception is propagated to the try statement during execution of the try [block](#_Trm00038):
  + The catch clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located:
    - If the matching catch clause declares an [exception variable](#_Trm00294), the exception [object](#_Trm00173) is assigned to the [exception variable](#_Trm00294).
    - Control is transferred to the matching catch [block](#_Trm00038).
    - When and if control reaches the [end point](#_Trm00264) of the catch [block](#_Trm00038):
      * If the try statement has a finally [block](#_Trm00038), the finally [block](#_Trm00038) is executed.
      * Control is transferred to the [end point](#_Trm00264) of the try statement.
    - If an exception is propagated to the try statement during execution of the catch [block](#_Trm00038):
      * If the try statement has a finally [block](#_Trm00038), the finally [block](#_Trm00038) 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](#_Trm00038), the finally [block](#_Trm00038) is executed.
    - The exception is propagated to the next enclosing try statement.

The [statements](#_Trm00037) of a finally [block](#_Trm00038) 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](#_Trm00038), and is not caught within the same finally [block](#_Trm00038), the exception is propagated to the next enclosing try statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the throw statement ([§8.9.5](#_Toc00371)).

The try [block](#_Trm00038) of a try statement is [reachable](#_Trm00265) if the try statement is [reachable](#_Trm00265).

A catch [block](#_Trm00038) of a try statement is [reachable](#_Trm00265) if the try statement is [reachable](#_Trm00265).

The finally [block](#_Trm00038) of a try statement is [reachable](#_Trm00265) if the try statement is [reachable](#_Trm00265).

The [end point](#_Trm00264) of a try statement is [reachable](#_Trm00265) if both of the following are true:

* The [end point](#_Trm00264) of the try [block](#_Trm00038) is [reachable](#_Trm00265) or the [end point](#_Trm00264) of at least one catch [block](#_Trm00038) is [reachable](#_Trm00265).
* If a finally [block](#_Trm00038) is present, the [end point](#_Trm00264) of the finally [block](#_Trm00038) is [reachable](#_Trm00265).

## The checked and unchecked [statements](#_Trm00037)

The checked and unchecked [statements](#_Trm00037) are used to control the ***overflow checking context*** for integral-type arithmetic operations and [conversion](#_Trm00196)s.

checked\_statement:  
 | 'checked' block  
 ;  
  
unchecked\_statement:  
 | 'unchecked' block  
 ;

The checked statement causes all expressions in the [block](#_Grm00071) to be evaluated in a checked context, and the unchecked statement causes all expressions in the [block](#_Grm00071) to be evaluated in an unchecked context.

The checked and unchecked [statements](#_Trm00037) are precisely equivalent to the checked and unchecked [operator](#_Trm00090)s ([§7.6.12](#_Toc00278)), except that they operate on [block](#_Trm00038)s instead of expressions.

## The lock statement

The lock statement obtains the mutual-exclusion lock for a given [object](#_Trm00173), executes a statement, and then releases the lock.

lock\_statement:  
 | 'lock' '(' expression ')' embedded\_statement  
 ;

The expression of a lock statement must denote a [value](#_Trm00209) of a type known to be a [reference\_type](#_Grm00030). No [implicit](#_Trm00197) [boxing](#_Trm00029) [conversion](#_Trm00196) ([§6.1.7](#_Toc00175)) 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](#_Trm00209) of a [value\_type](#_Grm00029).

A lock statement of the form

lock (x) ...

where x is an expression of a [reference\_type](#_Grm00030), 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 [block](#_Trm00038)ed from obtaining the lock until the lock is released.

Locking System.Type [object](#_Trm00173)s 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](#_Trm00173). 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](#_Trm00056) named Dispose. Code that is using a [resource](#_Trm00296) can call Dispose to indicate that the [resource](#_Trm00296) is no longer needed. If Dispose is not called, then automatic disposal [event](#_Trm00088)ually occurs as a con[sequence](#_Trm00259) of garbage collection.

If the form of [resource\_acquisition](#_Grm00096) is [local\_variable\_declaration](#_Grm00076) then the type of the [local\_variable\_declaration](#_Grm00076) must be either dynamic or a type that can be [implicit](#_Trm00197)ly converted to System.IDisposable. If the form of [resource\_acquisition](#_Grm00096) is [expression](#_Grm00067) then this expression must be [implicit](#_Trm00197)ly convertible to System.IDisposable.

Local [variables](#_Trm00031) declared in a [resource\_acquisition](#_Grm00096) are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these [local variable](#_Trm00193)s (via assignment or the ++ and -- [operator](#_Trm00090)s) , take the address of them, or pass them as ref or out [parameters](#_Trm00059).

A using statement is translated into three parts: acquisition, usage, and disposal. Usage of the [resource](#_Trm00296) is [implicit](#_Trm00197)ly enclosed in a try statement that includes a finally clause. This finally clause disposes of the [resource](#_Trm00296). If a null [resource](#_Trm00296) is acquired, then no call to Dispose is made, and no exception is thrown. If the [resource](#_Trm00296) is of type dynamic it is dynamically converted through an [implicit](#_Trm00197) dynamic [conversion](#_Trm00196) ([§6.1.8](#_Toc00176)) to IDisposable during acquisition in order to ensure that the [conversion](#_Trm00196) 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](#_Trm00169), the expansion is

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 ((IDisposable)resource).Dispose();  
 }  
}

Otherwise, when ResourceType is a nullable [value](#_Trm00209) 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 in[accessible](#_Trm00138) in, and in[visible](#_Trm00152) 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 [implicit](#_Trm00197)ly the [compile-time type](#_Trm00074) of the expression, if it has one. Otherwise the [interface](#_Trm00102) IDisposable itself is used as the ResourceType. The resource variable is in[accessible](#_Trm00138) in, and in[visible](#_Trm00152) to, the embedded statement.

When a [resource\_acquisition](#_Grm00096) takes the form of a [local\_variable\_declaration](#_Grm00076), it is possible to acquire multiple [resource](#_Trm00296)s 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](#_Trm00259) of [nested](#_Trm00143) using [statements](#_Trm00037):

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](#_Trm00102), the example can use using [statements](#_Trm00037) 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](#_Trm00038) ([§8.2](#_Toc00350)) to yield a [value](#_Trm00209) to the enumerator [object](#_Trm00173) ([§10.14.4](#_Toc00487)) or enumerable [object](#_Trm00173) ([§10.14.5](#_Toc00491)) 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](#_Trm00117). 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](#_Grm00116), [operator\_body](#_Grm00122) or [accessor\_body](#_Grm00119)
* It is a compile-time error for a yield statement (of either form) to appear inside an [anonymous function](#_Trm00253).
* 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](#_Trm00037).

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](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) must exist from the type of the expression in the yield return statement to the yield type ([§10.14.3](#_Toc00486)) of the iterator.

A yield return statement is executed as follows:

* The expression given in the statement is evaluated, [implicit](#_Trm00197)ly converted to the yield type, and assigned to the Current property of the enumerator [object](#_Trm00173).
* Execution of the iterator [block](#_Trm00038) is suspended. If the yield return statement is within one or more try [block](#_Trm00038)s, the associated finally [block](#_Trm00038)s are not executed at this time.
* The MoveNext [method](#_Trm00056) of the enumerator [object](#_Trm00173) returns true to its caller, indicating that the enumerator [object](#_Trm00173) successfully advanced to the next item.

The next call to the enumerator [object](#_Trm00173)'s MoveNext [method](#_Trm00056) resumes execution of the iterator [block](#_Trm00038) 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 [block](#_Trm00038)s with associated finally [block](#_Trm00038)s, control is initially transferred to the finally [block](#_Trm00038) of the innermost try statement. When and if control reaches the [end point](#_Trm00264) of a finally [block](#_Trm00038), control is transferred to the finally [block](#_Trm00038) of the next enclosing try statement. This process is repeated until the finally [block](#_Trm00038)s of all enclosing try [statements](#_Trm00037) have been executed.
* Control is returned to the caller of the iterator [block](#_Trm00038). This is either the MoveNext [method](#_Trm00056) or Dispose [method](#_Trm00056) of the enumerator [object](#_Trm00173).

Because a yield break statement unconditionally transfers control elsewhere, the [end point](#_Trm00264) of a yield break statement is never [reachable](#_Trm00265).

# Namespaces

C# [program](#_Trm00109)s are organized using [namespaces](#_Trm00010). Namespaces are used both as an "internal" organization system for a [program](#_Trm00109), and as an "external" organization system—a way of presenting [program](#_Trm00109) [elements](#_Trm00094) that are exposed to other [program](#_Trm00109)s.

Using directives ([§9.4](#_Toc00381)) are provided to facilitate the use of [namespaces](#_Trm00010).

## Compilation units

A [compilation\_unit](#_Grm00098) defines the overall structure of a source file. A compilation unit consists of zero or more [using\_directive](#_Grm00101)s followed by zero or more [global\_attributes](#_Grm00147) followed by zero or more [namespace\_member\_declaration](#_Grm00104)s.

compilation\_unit:  
 | extern\_alias\_directive\* using\_directive\* global\_attributes? namespace\_member\_declaration\*  
 ;

A C# [program](#_Trm00109) consists of one or more [compilation units](#_Trm00111), each contained in a separate source file. When a C# [program](#_Trm00109) is compiled, all of the [compilation units](#_Trm00111) are processed together. Thus, [compilation units](#_Trm00111) can depend on each other, possibly in a circular fashion.

The [using\_directive](#_Grm00101)s of a compilation unit affect the [global\_attributes](#_Grm00147) and [namespace\_member\_declaration](#_Grm00104)s of that compilation unit, but have no effect on other [compilation units](#_Trm00111).

The [global\_attributes](#_Grm00147) ([§17](#_Toc00567)) of a compilation unit permit the specification of [attributes](#_Trm00108) for the [target](#_Trm00290) assembly and module. Assemblies and modules act as physical containers for [types](#_Trm00011). An assembly may consist of several physically separate modules.

The [namespace\_member\_declaration](#_Grm00104)s of each compilation unit of a [program](#_Trm00109) contribute [members](#_Trm00012) to a single [declaration space](#_Trm00130) called the [global namespace](#_Trm00137). For example:

File A.cs:

class A {}

File B.cs:

class B {}

The two [compilation units](#_Trm00111) contribute to the single [global namespace](#_Trm00137), in this case declaring two classes with the [fully qualified name](#_Trm00153)s A and B. Because the two [compilation units](#_Trm00111) contribute to the same [declaration space](#_Trm00130), it would have been an error if each contained a declaration of a member with the same name.

## Namespace declarations

A [namespace\_declaration](#_Grm00099) consists of the [keyword](#_Trm00117) 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](#_Grm00099) may occur as a [top-level](#_Trm00142) declaration in a [compilation\_unit](#_Grm00098) or as a member declaration within another [namespace\_declaration](#_Grm00099). When a [namespace\_declaration](#_Grm00099) occurs as a [top-level](#_Trm00142) declaration in a [compilation\_unit](#_Grm00098), the namespace becomes a member of the [global namespace](#_Trm00137). When a [namespace\_declaration](#_Grm00099) occurs within another [namespace\_declaration](#_Grm00099), 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 [implicit](#_Trm00197)ly public and the declaration of a namespace cannot include any access modifiers.

Within a [namespace\_body](#_Grm00099), the optional [using\_directive](#_Grm00101)s import the names of other [namespaces](#_Trm00010) and [types](#_Trm00011), allowing them to be referenced directly instead of through qualified names. The optional [namespace\_member\_declaration](#_Grm00104)s contribute [members](#_Trm00012) to the [declaration space](#_Trm00130) of the namespace. Note that all [using\_directive](#_Grm00101)s must appear before any member declarations.

The [qualified\_identifier](#_Grm00099) of a [namespace\_declaration](#_Grm00099) may be a single identifier or a [sequence](#_Trm00259) of identifiers separated by "." tokens. The latter form permits a [program](#_Trm00109) to define a [nested](#_Trm00143) 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](#_Trm00153) contribute to the same [declaration space](#_Trm00130) ([§3.3](#_Toc00067)). In the example

namespace N1.N2  
{  
 class A {}  
}  
  
namespace N1.N2  
{  
 class B {}  
}

the two namespace declarations above contribute to the same [declaration space](#_Trm00130), in this case declaring two classes with the [fully qualified name](#_Trm00153)s N1.N2.A and N1.N2.B. Because the two declarations contribute to the same [declaration space](#_Trm00130), it would have been an error if each contained a declaration of a member with the same name.

## Extern aliases

An [extern\_alias\_directive](#_Grm00100) 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](#_Trm00109) and applies also to [nested](#_Trm00143) [namespaces](#_Trm00010) of the aliased namespace.

extern\_alias\_directive:  
 | 'extern' 'alias' identifier ';'  
 ;

The [scope](#_Trm00148) of an [extern\_alias\_directive](#_Grm00100) extends over the [using\_directive](#_Grm00101)s, [global\_attributes](#_Grm00147) and [namespace\_member\_declaration](#_Grm00104)s of its immediately containing compilation unit or namespace body.

Within a compilation unit or namespace body that contains an [extern\_alias\_directive](#_Grm00100), the identifier introduced by the [extern\_alias\_directive](#_Grm00100) can be used to reference the aliased namespace. It is a compile-time error for the [identifier](#_Grm00007) to be the word global.

An [extern\_alias\_directive](#_Grm00100) makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130). In other words, an [extern\_alias\_directive](#_Grm00100) is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.

The following [program](#_Trm00109) 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](#_Trm00109) declares the existence of the extern aliases X and Y, but the actual definitions of the aliases are external to the [program](#_Trm00109). 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](#_Trm00109) declares an extern alias for which no external definition is provided.

## Using directives

***Using directives*** facilitate the use of [namespaces](#_Trm00010) and [types](#_Trm00011) [defined](#_Trm00121) in other [namespaces](#_Trm00010). [Using directives](#_Trm00297) impact the name resolution process of [namespace\_or\_type\_name](#_Grm00027)s ([§3.8](#_Toc00086)) and [simple\_name](#_Grm00036)s ([§7.6.2](#_Toc00254)), but unlike declarations, using directives do not contribute new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130)s of the [compilation units](#_Trm00111) or [namespaces](#_Trm00010) within which they are used.

using\_directive:  
 | using\_alias\_directive  
 | using\_namespace\_directive  
 ;

A [using\_alias\_directive](#_Grm00102) ([§9.4.1](#_Toc00382)) introduces an alias for a namespace or type.

A [using\_namespace\_directive](#_Grm00103) ([§9.4.2](#_Toc00383)) imports the type [members](#_Trm00012) of a namespace.

The [scope](#_Trm00148) of a [using\_directive](#_Grm00101) extends over the [namespace\_member\_declaration](#_Grm00104)s of its immediately containing compilation unit or namespace body. The [scope](#_Trm00148) of a [using\_directive](#_Grm00101) specifically does not include its peer [using\_directive](#_Grm00101)s. Thus, peer [using\_directive](#_Grm00101)s do not affect each other, and the order in which they are written is insignificant.

### Using alias directives

A [using\_alias\_directive](#_Grm00102) 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](#_Grm00102), the identifier introduced by the [using\_alias\_directive](#_Grm00102) 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](#_Grm00007) of a [using\_alias\_directive](#_Grm00102) must be unique within the [declaration space](#_Trm00130) of the compilation unit or namespace that immediately contains the [using\_alias\_directive](#_Grm00102). 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](#_Grm00102) to use that identifier. Likewise, it is a compile-time error for two or more [using\_alias\_directive](#_Grm00102)s in the same compilation unit or namespace body to declare aliases by the same name.

A [using\_alias\_directive](#_Grm00102) makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130). In other words, a [using\_alias\_directive](#_Grm00102) 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](#_Trm00148) of the [using\_alias\_directive](#_Grm00102) 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](#_Grm00102) 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](#_Trm00012), names introduced by [using\_alias\_directive](#_Grm00102)s are [hidden](#_Trm00150) by similarly named [members](#_Trm00012) in [nested](#_Trm00143) [scope](#_Trm00148)s. 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](#_Grm00102)s are written has no significance, and resolution of the [namespace\_or\_type\_name](#_Grm00027) referenced by a [using\_alias\_directive](#_Grm00102) is not affected by the [using\_alias\_directive](#_Grm00102) itself or by other [using\_directive](#_Grm00101)s in the immediately containing compilation unit or namespace body. In other words, the [namespace\_or\_type\_name](#_Grm00027) of a [using\_alias\_directive](#_Grm00102) is resolved as if the immediately containing compilation unit or namespace body had no [using\_directive](#_Grm00101)s. A [using\_alias\_directive](#_Grm00102) may however be affected by [extern\_alias\_directive](#_Grm00100)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](#_Grm00102) results in a compile-time error because it is not affected by the first [using\_alias\_directive](#_Grm00102). The first [using\_alias\_directive](#_Grm00102) does not result in an error since the [scope](#_Trm00148) of the extern alias E includes the [using\_alias\_directive](#_Grm00102).

A [using\_alias\_directive](#_Grm00102) can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type [nested](#_Trm00143) 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](#_Trm00153) is N1.N2.A.

Using aliases can name a closed [constructed type](#_Trm00178), but cannot name an [unbound generic type](#_Trm00176) declaration without supplying type [arguments](#_Trm00062). 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](#_Grm00103) imports the [types](#_Trm00011) 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](#_Grm00103), the [types](#_Trm00011) 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](#_Trm00012) of N1.N2 are directly available, and thus class N3.B derives from class N1.N2.A.

A [using\_namespace\_directive](#_Grm00103) imports the [types](#_Trm00011) contained in the given namespace, but specifically does not import [nested](#_Trm00143) [namespaces](#_Trm00010). In the example

namespace N1.N2  
{  
 class A {}  
}  
  
namespace N3  
{  
 using N1;  
  
 class B: N2.A {} // Error, N2 unknown  
}

the [using\_namespace\_directive](#_Grm00103) imports the [types](#_Trm00011) contained in N1, but not the [namespaces](#_Trm00010) [nested](#_Trm00143) in N1. Thus, the reference to N2.A in the declaration of B results in a compile-time error because no [members](#_Trm00012) named N2 are in [scope](#_Trm00148).

Unlike a [using\_alias\_directive](#_Grm00102), a [using\_namespace\_directive](#_Grm00103) may import [types](#_Trm00011) whose identifiers are already [defined](#_Trm00121) within the enclosing compilation unit or namespace body. In effect, names imported by a [using\_namespace\_directive](#_Grm00103) are [hidden](#_Trm00150) by similarly named [members](#_Trm00012) 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 imported by [using\_namespace\_directive](#_Grm00103)s in the same compilation unit or namespace body contain [types](#_Trm00011) by the same name, [references](#_Trm00160) to that 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](#_Trm00160) to A, or by introducing a [using\_alias\_directive](#_Grm00102) that picks a particular A. For example:

namespace N3  
{  
 using N1;  
  
 using N2;  
  
 using A = N1.A;  
  
 class B: A {} // A means N1.A  
}

Like a [using\_alias\_directive](#_Grm00102), a [using\_namespace\_directive](#_Grm00103) does not contribute any new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130) of the compilation unit or namespace, but rather affects only the compilation unit or namespace body in which it appears.

The [namespace\_name](#_Grm00027) referenced by a [using\_namespace\_directive](#_Grm00103) is resolved in the same way as the [namespace\_or\_type\_name](#_Grm00027) referenced by a [using\_alias\_directive](#_Grm00102). Thus, [using\_namespace\_directive](#_Grm00103)s in the same compilation unit or namespace body do not affect each other and can be written in any order.

## Namespace [members](#_Trm00012)

A [namespace\_member\_declaration](#_Grm00104) is either a [namespace\_declaration](#_Grm00099) ([§9.2](#_Toc00379)) or a [type\_declaration](#_Grm00105) ([§9.6](#_Toc00385)).

namespace\_member\_declaration:  
 | namespace\_declaration  
 | type\_declaration  
 ;

A compilation unit or a namespace body can contain [namespace\_member\_declaration](#_Grm00104)s, and such declarations contribute new [members](#_Trm00012) to the underlying [declaration space](#_Trm00130) of the containing compilation unit or namespace body.

## Type declarations

A [type\_declaration](#_Grm00105) is a [class\_declaration](#_Grm00107) ([§10.1](#_Toc00389)), a [struct\_declaration](#_Grm00126) ([§11.1](#_Toc00498)), an [interface\_declaration](#_Grm00133) ([§13.1](#_Toc00528)), an [enum\_declaration](#_Grm00143) ([§14.1](#_Toc00552)), or a [delegate\_declaration](#_Grm00146) ([§15.1](#_Toc00558)).

type\_declaration:  
 | class\_declaration  
 | struct\_declaration  
 | interface\_declaration  
 | enum\_declaration  
 | delegate\_declaration  
 ;

A [type\_declaration](#_Grm00105) can occur as a [top-level](#_Trm00142) 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](#_Trm00142) declaration in a compilation unit, the [fully qualified name](#_Trm00153) 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](#_Trm00153) of the newly declared type is N.T, where N is the [fully qualified name](#_Trm00153) of the containing namespace, class, or struct.

A type declared within a class or struct is called a [nested](#_Trm00143) type ([§10.3.8](#_Toc00418)).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place ([§3.5.1](#_Toc00077)):

* Types declared in [compilation units](#_Trm00111) or [namespaces](#_Trm00010) 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](#_Trm00092) 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](#_Trm00011) and [members](#_Trm00012). The [namespace alias qualifier](#_Trm00298) 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](#_Grm00106) is [defined](#_Trm00121) as follows:

qualified\_alias\_member:  
 | identifier '::' identifier type\_argument\_list?  
 ;

A [qualified\_alias\_member](#_Grm00106) can be used as a [namespace\_or\_type\_name](#_Grm00027) ([§3.8](#_Toc00086)) or as the left operand in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)).

A [qualified\_alias\_member](#_Grm00106) 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](#_Grm00106) is determined as follows:

* If N is the identifier global, then the [global namespace](#_Trm00137) is searched for I:
  + If the [global namespace](#_Trm00137) contains a namespace named I and K is zero, then the [qualified\_alias\_member](#_Grm00106) refers to that namespace.
  + Otherwise, if the [global namespace](#_Trm00137) contains a non-generic type named I and K is zero, then the [qualified\_alias\_member](#_Grm00106) refers to that type.
  + Otherwise, if the [global namespace](#_Trm00137) contains a type named I that has K type [parameters](#_Trm00059), then the [qualified\_alias\_member](#_Grm00106) refers to that type constructed with the given type [arguments](#_Trm00062).
  + Otherwise, the [qualified\_alias\_member](#_Grm00106) is un[defined](#_Trm00121) and a compile-time error occurs.
* Otherwise, starting with the namespace declaration ([§9.2](#_Toc00379)) immediately containing the [qualified\_alias\_member](#_Grm00106) (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the [qualified\_alias\_member](#_Grm00106), the following steps are evaluated until an entity is located:
  + If the namespace declaration or compilation unit contains a [using\_alias\_directive](#_Grm00102) that associates N with a type, then the [qualified\_alias\_member](#_Grm00106) is un[defined](#_Trm00121) and a compile-time error occurs.
  + Otherwise, if the namespace declaration or compilation unit contains an [extern\_alias\_directive](#_Grm00100) or [using\_alias\_directive](#_Grm00102) 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](#_Grm00106) 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](#_Grm00106) refers to that type.
    - Otherwise, if the namespace associated with N contains a type named I that has K type [parameters](#_Trm00059), then the [qualified\_alias\_member](#_Grm00106) refers to that type constructed with the given type [arguments](#_Trm00062).
    - Otherwise, the [qualified\_alias\_member](#_Grm00106) is un[defined](#_Trm00121) and a compile-time error occurs.
* Otherwise, the [qualified\_alias\_member](#_Grm00106) is un[defined](#_Trm00121) and a compile-time error occurs.

Note that using the [namespace alias qualifier](#_Trm00298) with an alias that [references](#_Trm00160) a type causes a compile-time error. Also note that if the identifier N is global, then lookup is performed in the [global namespace](#_Trm00137), 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](#_Trm00130) 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](#_Trm00148). 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](#_Trm00044)

A class is a data structure that may contain data [members](#_Trm00012) (constants and fields), [function members](#_Trm00079) ([method](#_Trm00056)s, properties, [event](#_Trm00088)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance](#_Trm00172) constructors, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s), and [nested](#_Trm00143) [types](#_Trm00011). Class [types](#_Trm00011) support [inheritance](#_Trm00047), a mechanism whereby a derived class can extend and specialize a base class.

## Class declarations

A [class\_declaration](#_Grm00107) is a [type\_declaration](#_Grm00105) ([§9.6](#_Toc00385)) 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](#_Grm00107) consists of an optional set of [attributes](#_Grm00147) ([§17](#_Toc00567)), followed by an optional set of [class\_modifier](#_Grm00108)s ([§10.1.1](#_Toc00390)), followed by an optional partial modifier, followed by the [keyword](#_Trm00117) class and an [identifier](#_Grm00007) that names the class, followed by an optional [type\_parameter\_list](#_Grm00109) ([§10.1.3](#_Toc00395)), followed by an optional [class\_base](#_Grm00110) specification ([§10.1.4](#_Toc00396)) , followed by an optional set of [type\_parameter\_constraints\_clause](#_Grm00111)s ([§10.1.5](#_Toc00399)), followed by a [class\_body](#_Grm00112) ([§10.1.6](#_Toc00400)), optionally followed by a semicolon.

A class declaration cannot supply [type\_parameter\_constraints\_clause](#_Grm00111)s unless it also supplies a [type\_parameter\_list](#_Grm00109).

A class declaration that supplies a [type\_parameter\_list](#_Grm00109) is a ***generic class declaration***. Additionally, any class [nested](#_Trm00143) inside a [generic class declaration](#_Trm00299) or a generic struct declaration is itself a [generic class declaration](#_Trm00299), since type [parameters](#_Trm00059) for the containing type must be supplied to create a [constructed type](#_Trm00178).

### Class modifiers

A [class\_declaration](#_Grm00107) may optionally include a [sequence](#_Trm00259) 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](#_Trm00143) classes. It specifies that the class [hide](#_Trm00132)s an [inherited](#_Trm00136) member by the same name, as described in [§10.3.4](#_Toc00414). It is a compile-time error for the new modifier to appear on a class declaration that is not a [nested](#_Trm00143) 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 ([§3.5.1](#_Toc00077)).

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](#_Trm00076) class differs from a non-[abstract](#_Trm00076) class in the following ways:

* An [abstract](#_Trm00076) class cannot be [instantiated](#_Trm00256) directly, and it is a compile-time error to use the new [operator](#_Trm00090) on an [abstract](#_Trm00076) class. While it is possible to have [variables](#_Trm00031) and [value](#_Trm00209)s whose [compile-time type](#_Trm00074)s are [abstract](#_Trm00076), such [variables](#_Trm00031) and [value](#_Trm00209)s will necessarily either be null or contain [references](#_Trm00160) to [instance](#_Trm00172)s of non-[abstract](#_Trm00076) classes derived from the [abstract](#_Trm00076) [types](#_Trm00011).
* An [abstract](#_Trm00076) class is permitted (but not required) to contain [abstract](#_Trm00076) [members](#_Trm00012).
* An [abstract](#_Trm00076) class cannot be sealed.

When a non-[abstract](#_Trm00076) class is derived from an [abstract](#_Trm00076) class, the non-[abstract](#_Trm00076) class must include actual implementations of all [inherited](#_Trm00136) [abstract](#_Trm00076) [members](#_Trm00012), thereby overriding those [abstract](#_Trm00076) [members](#_Trm00012). 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](#_Trm00076) class A introduces an [abstract](#_Trm00076) [method](#_Trm00056) F. Class B introduces an additional [method](#_Trm00056) G, but since it doesn't provide an implementation of F, B must also be declared [abstract](#_Trm00076). Class C overrides F and provides an actual implementation. Since there are no [abstract](#_Trm00076) [members](#_Trm00012) in C, C is permitted (but not required) to be non-[abstract](#_Trm00076).

#### Sealed classes

The sealed modifier is used to pr[event](#_Trm00088) 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](#_Trm00076) class.

The sealed modifier is primarily used to pr[event](#_Trm00088) unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any [derived classes](#_Trm00049), it is possible to transform virtual function member invocations on sealed class [instance](#_Trm00172)s into non-virtual invocations.

#### Static classes

The static modifier is used to mark the class being declared as a ***static class***. A [static class](#_Trm00300) cannot be [instantiated](#_Trm00256), cannot be used as a type and can contain only static [members](#_Trm00012). Only a [static class](#_Trm00300) can contain declarations of extension [method](#_Trm00056)s ([§10.6.9](#_Toc00454)).

A [static class](#_Trm00300) declaration is subject to the following restrictions:

* A [static class](#_Trm00300) may not include a sealed or abstract modifier. Note, however, that since a [static class](#_Trm00300) cannot be [instantiated](#_Trm00256) or derived from, it behaves as if it was both sealed and [abstract](#_Trm00076).
* A [static class](#_Trm00300) may not include a [class\_base](#_Grm00110) specification ([§10.1.4](#_Toc00396)) and cannot [explicit](#_Trm00198)ly specify a base class or a list of implemented [interface](#_Trm00102)s. A [static class](#_Trm00300) [implicit](#_Trm00197)ly inherits from type object.
* A [static class](#_Trm00300) can only contain static [members](#_Trm00012) ([§10.3.7](#_Toc00417)). Note that constants and [nested](#_Trm00143) [types](#_Trm00011) are classified as static [members](#_Trm00012).
* A [static class](#_Trm00300) cannot have [members](#_Trm00012) with protected or protected internal [declared accessibility](#_Trm00140).

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

A [static class](#_Trm00300) has no [instance](#_Trm00172) constructors. It is not possible to declare an [instance](#_Trm00172) constructor in a [static class](#_Trm00300), and no default [instance](#_Trm00172) constructor ([§10.11.4](#_Toc00478)) is provided for a [static class](#_Trm00300).

The [members](#_Trm00012) of a [static class](#_Trm00300) are not automatically static, and the member declarations must [explicit](#_Trm00198)ly include a static modifier (except for constants and [nested](#_Trm00143) [types](#_Trm00011)). When a class is [nested](#_Trm00143) within a static outer class, the [nested](#_Trm00143) class is not a [static class](#_Trm00300) unless it [explicit](#_Trm00198)ly includes a static modifier.

**Referencing static class types**

A [namespace\_or\_type\_name](#_Grm00027) ([§3.8](#_Toc00086)) is permitted to reference a [static class](#_Trm00300) if

* The [namespace\_or\_type\_name](#_Grm00027) is the T in a [namespace\_or\_type\_name](#_Grm00027) of the form T.I, or
* The [namespace\_or\_type\_name](#_Grm00027) is the T in a [typeof\_expression](#_Grm00050) ([§7.5.1](#_Toc00224)1) of the form typeof(T).

A [primary\_expression](#_Grm00035) ([§7.5](#_Toc00223)) is permitted to reference a [static class](#_Trm00300) if

* The [primary\_expression](#_Grm00035) is the E in a [member\_access](#_Grm00038) ([§7.5.4](#_Toc00249)) of the form E.I.

In any other context it is a compile-time error to reference a [static class](#_Trm00300). For example, it is an error for a [static class](#_Trm00300) to be used as a base class, a constituent type ([§10.3.8](#_Toc00418)) of a member, a generic type argument, or a type parameter constraint. Likewise, a [static class](#_Trm00300) cannot be used in an [array](#_Trm00093) type, a pointer type, a new expression, a cast expression, an is expression, an as expression, a sizeof expression, or a [default value](#_Trm00164) expression.

### Partial modifier

The partial modifier is used to indicate that this [class\_declaration](#_Grm00107) is a partial type declaration. Multiple partial [type declarations](#_Trm00028) with the same name within an enclosing namespace or type declaration combine to form one type declaration, following the rules specified in [§10.2](#_Toc00401).

Having the declaration of a class distributed over separate segments of [program](#_Trm00109) text can be useful if these segments are produced or maintained in different contexts. For [instance](#_Trm00172), one part of a class declaration may be machine generated, whereas the other is manually authored. Textual separation of the two pr[event](#_Trm00088)s updates by one from conflicting with updates by the other.

### Type [parameters](#_Trm00059)

A type parameter is a simple identifier that denotes a placeholder for a type argument supplied to create a [constructed type](#_Trm00178). A type parameter is a formal placeholder for a type that will be supplied later. By constrast, a type argument ([§4.4.1](#_Toc00114)) is the actual type that is substituted for the type parameter when a [constructed type](#_Trm00178) 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](#_Trm00130) ([§3.3](#_Toc00067)) 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](#_Grm00110) specification, which defines the direct base class of the class and the [interface](#_Trm00102)s ([§13](#_Toc00527)) 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 ([§4.4](#_Toc00113)). A base class cannot be a type parameter on its own, though it can involve the type [parameters](#_Trm00059) that are in [scope](#_Trm00148).

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

#### Base classes

When a [class\_type](#_Grm00030) is included in the [class\_base](#_Grm00110), it specifies the direct base class of the class being declared. If a class declaration has no [class\_base](#_Grm00110), or if the [class\_base](#_Grm00110) lists only [interface](#_Trm00102) [types](#_Trm00011), the direct base class is assumed to be object. A class inherits [members](#_Trm00012) from its direct base class, as described in [§10.3.3](#_Toc00413).

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 [explicit](#_Trm00198)ly specify a direct base class, its direct base class is [implicit](#_Trm00197)ly object.

For a constructed class type, if a base class is specified in the [generic class declaration](#_Trm00299), the base class of the [constructed type](#_Trm00178) is obtained by substituting, for each [type\_parameter](#_Grm00032) in the base class declaration, the corresponding [type\_argument](#_Grm00031) of the [constructed type](#_Trm00178). Given the [generic class declaration](#_Trm00299)s

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

the base class of the [constructed type](#_Trm00178) G<int> would be B<string,int[]>.

The direct base class of a class type must be at least as [accessible](#_Trm00138) as the class type itself ([§3.5.2](#_Toc00078)). 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](#_Trm00011): System.Array, System.Delegate, System.MulticastDelegate, System.Enum, or System.ValueType. Furthermore, a [generic class declaration](#_Trm00299) 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 [§3.8](#_Toc00086)) C is not considered to have a member B.

The [base classes](#_Trm00050) of a class type are the direct base class and its [base classes](#_Trm00050). In other words, the set of [base classes](#_Trm00050) is the transitive closure of the direct base class relationship. Referring to the example above, the [base classes](#_Trm00050) 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](#_Trm00050) 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](#_Trm00143) (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](#_Trm00143) 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](#_Grm00110) specification may include a list of [interface](#_Trm00102) [types](#_Trm00011), in which case the class is said to directly implement the given [interface](#_Trm00102) [types](#_Trm00011). Interface implementations are discussed further in [§13.4](#_Toc00543).

### Type parameter constraints

Generic type and [method](#_Trm00056) declarations can optionally specify type parameter constraints by including [type\_parameter\_constraints\_clause](#_Grm00111)s.

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](#_Grm00111) 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](#_Trm00117).

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](#_Grm00032) or [interface\_type](#_Grm00030).

The [reference type constraint](#_Trm00304) specifies that a type argument used for the type parameter must be a reference type. All [class types](#_Trm00024), [interface](#_Trm00102) [types](#_Trm00011), [delegate type](#_Trm00107)s, [array](#_Trm00093) [types](#_Trm00011), and type [parameters](#_Trm00059) known to be a reference type (as [defined](#_Trm00121) below) satisfy this constraint.

The [value](#_Trm00209) type constraint specifies that a type argument used for the type parameter must be a [non-nullable value type](#_Trm00169). All non-nullable [struct types](#_Trm00022), [enum type](#_Trm00105)s, and type [parameters](#_Trm00059) having the [value](#_Trm00209) type constraint satisfy this constraint. Note that although classified as a [value](#_Trm00209) type, a nullable type ([§4.1.10](#_Toc00101)) does not satisfy the [value](#_Trm00209) type constraint. A type parameter having the [value](#_Trm00209) type constraint cannot also have the [constructor\_constraint](#_Grm00111).

Pointer [types](#_Trm00011) are never allowed to be type [arguments](#_Trm00062) and are not considered to satisfy either the reference type or [value](#_Trm00209) type constraints.

If a constraint is a class type, an [interface](#_Trm00102) 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](#_Trm00178) or generic [method](#_Trm00056) 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 [§4.4.4](#_Toc00117).

A [class\_type](#_Grm00030) 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](#_Trm00011): System.Array, System.Delegate, System.Enum, or System.ValueType.
* The type must not be object. Because all [types](#_Trm00011) 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](#_Grm00030) constraint must satisfy the following rules:

* The type must be an [interface](#_Trm00102) 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](#_Trm00059) of the associated type or [method](#_Trm00056) declaration as part of a [constructed type](#_Trm00178), and can involve the type being declared.

Any class or [interface](#_Trm00102) type specified as a type parameter constraint must be at least as [accessible](#_Trm00138) ([§3.5.4](#_Toc00080)) as the generic type or [method](#_Trm00056) being declared.

A type specified as a [type\_parameter](#_Grm00032) 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](#_Trm00059), where dependency is a transitive relation [defined](#_Trm00121) 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](#_Trm00306) a type parameter T and T [depends on](#_Trm00306) 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](#_Trm00059). If type parameter S [depends on](#_Trm00306) type parameter T then:

* T must not have the [value](#_Trm00209) 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](#_Trm00059).
* If S has the [value](#_Trm00209) type constraint then T must not have a [class\_type](#_Grm00030) constraint.
* If S has a [class\_type](#_Grm00030) constraint A and T has a [class\_type](#_Grm00030) constraint B then there must be an identity [conversion](#_Trm00196) or [implicit](#_Trm00197) reference [conversion](#_Trm00196) from A to B or an [implicit](#_Trm00197) reference [conversion](#_Trm00196) from B to A.
* If S also [depends on](#_Trm00306) type parameter U and U has a [class\_type](#_Grm00030) constraint A and T has a [class\_type](#_Grm00030) constraint B then there must be an identity [conversion](#_Trm00196) or [implicit](#_Trm00197) reference [conversion](#_Trm00196) from A to B or an [implicit](#_Trm00197) reference [conversion](#_Trm00196) from B to A.

It is valid for S to have the [value](#_Trm00209) type constraint and T to have the [reference type constraint](#_Trm00304). Effectively this limits T to the [types](#_Trm00011) System.Object, System.ValueType, System.Enum, and any [interface](#_Trm00102) 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](#_Trm00090) to create [instance](#_Trm00172)s of the type ([§7.6.10.1](#_Toc00271)). Any type argument used for a type parameter with a constructor constraint must have a public parameterless constructor (this constructor [implicit](#_Trm00197)ly exists for any [value](#_Trm00209) type) or be a type parameter having the [value](#_Trm00209) type constraint or constructor constraint (see [§10.1.5](#_Toc00399) 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](#_Trm00059):

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](#_Trm00121) as follows:

* If T has no primary constraints or type parameter constraints, its [effective base class](#_Trm00308) is object.
* If T has the [value](#_Trm00209) type constraint, its [effective base class](#_Trm00308) is System.ValueType.
* If T has a [class\_type](#_Grm00030) constraint C but no [type\_parameter](#_Grm00032) constraints, its [effective base class](#_Trm00308) is C.
* If T has no [class\_type](#_Grm00030) constraint but has one or more [type\_parameter](#_Grm00032) constraints, its [effective base class](#_Trm00308) is the most [encompass](#_Trm00206)ed type ([§6.4.2](#_Toc00195)) in the set of effective [base classes](#_Trm00050) of its [type\_parameter](#_Grm00032) constraints. The consistency rules ensure that such a most [encompass](#_Trm00206)ed type exists.
* If T has both a [class\_type](#_Grm00030) constraint and one or more [type\_parameter](#_Grm00032) constraints, its [effective base class](#_Trm00308) is the most [encompass](#_Trm00206)ed type ([§6.4.2](#_Toc00195)) in the set consisting of the [class\_type](#_Grm00030) constraint of T and the effective [base classes](#_Trm00050) of its [type\_parameter](#_Grm00032) constraints. The consistency rules ensure that such a most [encompass](#_Trm00206)ed type exists.
* If T has the [reference type constraint](#_Trm00304) but no [class\_type](#_Grm00030) constraints, its [effective base class](#_Trm00308) is object.

For the purpose of these rules, if T has a constraint V that is a [value\_type](#_Grm00029), use instead the [most specific](#_Trm00204) base type of V that is a [class\_type](#_Grm00030). This can never happen in an [explicit](#_Trm00198)ly given constraint, but may occur when the constraints of a generic [method](#_Trm00056) are [implicit](#_Trm00197)ly [inherited](#_Trm00136) by an overriding [method](#_Trm00056) declaration or an [explicit](#_Trm00198) implementation of an [interface](#_Trm00102) [method](#_Trm00056).

These rules ensure that the [effective base class](#_Trm00308) is always a [class\_type](#_Grm00030).

The ***effective interface set*** of a type parameter T is [defined](#_Trm00121) as follows:

* If T has no [secondary\_constraints](#_Grm00111), its [effective interface set](#_Trm00309) is empty.
* If T has [interface\_type](#_Grm00030) constraints but no [type\_parameter](#_Grm00032) constraints, its [effective interface set](#_Trm00309) is its set of [interface\_type](#_Grm00030) constraints.
* If T has no [interface\_type](#_Grm00030) constraints but has [type\_parameter](#_Grm00032) constraints, its [effective interface set](#_Trm00309) is the union of the [effective interface set](#_Trm00309)s of its [type\_parameter](#_Grm00032) constraints.
* If T has both [interface\_type](#_Grm00030) constraints and [type\_parameter](#_Grm00032) constraints, its [effective interface set](#_Trm00309) is the union of its set of [interface\_type](#_Grm00030) constraints and the [effective interface set](#_Trm00309)s of its [type\_parameter](#_Grm00032) constraints.

A type parameter is ***known to be a reference type*** if it has the [reference type constraint](#_Trm00304) or its [effective base class](#_Trm00308) is not object or System.ValueType.

Values of a constrained type parameter type can be used to access the [instance](#_Trm00172) [members](#_Trm00012) implied by the constraints. In the example

interface IPrintable  
{  
 void Print();  
}  
  
class Printer<T> where T: IPrintable  
{  
 void PrintOne(T x) {  
 x.Print();  
 }  
}

the [method](#_Trm00056)s of IPrintable can be invoked directly on x because T is constrained to always implement IPrintable.

### Class body

The [class\_body](#_Grm00112) of a class defines the [members](#_Trm00012) of that class.

class\_body:  
 | '{' class\_member\_declaration\* '}'  
 ;

## Partial [types](#_Trm00011)

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](#_Trm00109).

A [class\_declaration](#_Grm00107), [struct\_declaration](#_Grm00126) or [interface\_declaration](#_Grm00133) represents a partial type declaration if it includes a partial modifier. partial is not a [keyword](#_Trm00117), and only acts as a modifier if it appears immediately before one of the [keyword](#_Trm00117)s class, struct or interface in a type declaration, or before the type void in a [method](#_Trm00056) 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](#_Trm00011) specifically do not allow already compiled [types](#_Trm00011) to be extended.

Nested [types](#_Trm00011) 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](#_Trm00143) 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](#_Trm00108) of a partial type are determined by combining, in an unspecified order, the [attributes](#_Trm00108) 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](#_Trm00059) 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 ([§3.5.1](#_Toc00077)).

If one or more partial declarations of a [nested](#_Trm00143) type include a new modifier, no warning is reported if the [nested](#_Trm00143) type [hide](#_Trm00132)s an [inherited](#_Trm00136) member ([§3.7.1.2](#_Toc00085)).

If one or more partial declarations of a class include an abstract modifier, the class is considered [abstract](#_Trm00076) ([§10.1.1.1](#_Toc00391)). Otherwise, the class is considered non-[abstract](#_Trm00076).

If one or more partial declarations of a class include a sealed modifier, the class is considered sealed ([§10.1.1.2](#_Toc00392)). Otherwise, the class is considered unsealed.

Note that a class cannot be both [abstract](#_Trm00076) and sealed.

When the unsafe modifier is used on a partial type declaration, only that particular part is considered an unsafe context ([§18.1](#_Toc00591)).

### Type [parameters](#_Trm00059) and constraints

If a generic type is declared in multiple parts, each part must state the type [parameters](#_Trm00059). Each part must have the same number of type [parameters](#_Trm00059), 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](#_Trm00059), 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](#_Trm00012). If no part of a partial generic type specifies type parameter constraints, the type [parameters](#_Trm00059) 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](#_Trm00059), 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 ([§10.1.4.1](#_Toc00397)).

### Base [interface](#_Trm00102)s

The set of base [interface](#_Trm00102)s for a type declared in multiple parts is the union of the base [interface](#_Trm00102)s specified on each part. A particular base [interface](#_Trm00102) may only be named once on each part, but it is permitted for multiple parts to name the same base [interface](#_Trm00102)(s). There must only be one implementation of the [members](#_Trm00012) of any given base [interface](#_Trm00102).

In the example

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

the set of base [interface](#_Trm00102)s for class C is IA, IB, and IC.

Typically, each part provides an implementation of the [interface](#_Trm00102)(s) declared on that part; however, this is not a requirement. A part may provide the implementation for an [interface](#_Trm00102) declared on a different part:

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

### Members

With the exception of partial [method](#_Trm00056)s ([§10.2.7](#_Toc00408)), the set of [members](#_Trm00012) of a type declared in multiple parts is simply the union of the set of [members](#_Trm00012) declared in each part. The bodies of all parts of the type declaration share the same [declaration space](#_Trm00130) ([§3.3](#_Toc00067)), and the [scope](#_Trm00148) of each member ([§3.7](#_Toc00082)) extends to the bodies of all the parts. The [accessibility domain](#_Trm00141) of any member always includes all the parts of the enclosing type; a private member declared in one part is freely [accessible](#_Trm00138) 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](#_Trm00012) 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](#_Trm00012) within a type declared in multiple parts is un[defined](#_Trm00121).

### Partial [method](#_Trm00056)s

Partial [method](#_Trm00056)s can be [defined](#_Trm00121) in one part of a type declaration and implemented in another. The implementation is optional; if no part implements the partial [method](#_Trm00056), the partial [method](#_Trm00056) declaration and all calls to it are removed from the type declaration resulting from the combination of the parts.

Partial [method](#_Trm00056)s cannot define access modifiers, but are [implicit](#_Trm00197)ly private. Their [return type](#_Trm00060) must be void, and their [parameters](#_Trm00059) cannot have the out modifier. The identifier partial is recognized as a special [keyword](#_Trm00117) in a [method](#_Trm00056) declaration only if it appears right before the void type; otherwise it can be used as a normal identifier. A partial [method](#_Trm00056) cannot [explicit](#_Trm00198)ly implement [interface](#_Trm00102) [method](#_Trm00056)s.

There are two kinds of partial [method](#_Trm00056) declarations: If the body of the [method](#_Trm00056) declaration is a semicolon, the declaration is said to be a ***defining partial method declaration***. If the body is given as a [block](#_Grm00071), 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](#_Trm00312) with a given [signature](#_Trm00061), and there can be only one [implementing partial method declaration](#_Trm00313) with a given [signature](#_Trm00061). If an [implementing partial method declaration](#_Trm00313) is given, a corresponding [defining partial method declaration](#_Trm00312) 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](#_Trm00056) name, number of type [parameters](#_Trm00059) and number of [parameters](#_Trm00059).
* Corresponding [parameters](#_Trm00059) in the declarations must have the same modifiers (although not necessarily in the same order) and the same [types](#_Trm00011) (modulo differences in type parameter names).
* Corresponding type [parameters](#_Trm00059) in the declarations must have the same constraints (modulo differences in type parameter names).

An [implementing partial method declaration](#_Trm00313) can appear in the same part as the corresponding [defining partial method declaration](#_Trm00312).

Only a defining partial [method](#_Trm00056) participates in [overload resolution](#_Trm00078). Thus, whether or not an implementing declaration is given, invocation expressions may resolve to invocations of the partial [method](#_Trm00056). Because a partial [method](#_Trm00056) always returns void, such invocation expressions will always be expression [statements](#_Trm00037). Furthermore, because a partial [method](#_Trm00056) is [implicit](#_Trm00197)ly private, such [statements](#_Trm00037) will always occur within one of the parts of the type declaration within which the partial [method](#_Trm00056) is declared.

If no part of a partial type declaration contains an implementing declaration for a given partial [method](#_Trm00056), 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](#_Trm00056) 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](#_Trm00056), the invocations of the partial [method](#_Trm00056)s are retained. The partial [method](#_Trm00056) gives rise to a [method](#_Trm00056) declaration similar to the [implementing partial method declaration](#_Trm00313) except for the following:

* The partial modifier is not included
* The [attributes](#_Trm00108) in the resulting [method](#_Trm00056) declaration are the combined [attributes](#_Trm00108) of the defining and the [implementing partial method declaration](#_Trm00313) in unspecified order. Duplicates are not removed.
* The [attributes](#_Trm00108) on the [parameters](#_Trm00059) of the resulting [method](#_Trm00056) declaration are the combined [attributes](#_Trm00108) of the corresponding [parameters](#_Trm00059) of the defining and the [implementing partial method declaration](#_Trm00313) in unspecified order. Duplicates are not removed.

If a defining declaration but not an implementing declaration is given for a partial [method](#_Trm00056) M, the following restrictions apply:

* It is a compile-time error to create a delegate to [method](#_Trm00056) ([§7.6.10.5](#_Toc00275)).
* It is a compile-time error to refer to M inside an [anonymous function](#_Trm00253) that is converted to an expression tree type ([§6.5.2](#_Toc00201)).
* [Expressions](#_Trm00032) occurring as part of an invocation of M do not affect the [definite assignment state](#_Trm00195) ([§5.3](#_Toc00132)), which can potentially lead to compile-time errors.
* M cannot be the [entry point](#_Trm00123) for an [application](#_Trm00124) ([§3.1](#_Toc00065)).

Partial [method](#_Trm00056)s 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 declaration](#_Trm00312)s 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 [method](#_Trm00056)s:

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](#_Trm00210)

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 ([§9.4](#_Toc00381)) may be present for each part. When interpreting simple names ([§7.5.2](#_Toc00227)) 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](#_Trm00012)

The [members](#_Trm00012) of a class consist of the [members](#_Trm00012) introduced by its [class\_member\_declaration](#_Grm00113)s and the [members](#_Trm00012) [inherited](#_Trm00136) 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](#_Trm00012) of a class type are divided into the following categories:

* Constants, which represent constant [value](#_Trm00209)s associated with the class ([§10.4](#_Toc00430)).
* Fields, which are the [variables](#_Trm00031) of the class ([§10.5](#_Toc00431)).
* Methods, which implement the computations and actions that can be performed by the class ([§10.6](#_Toc00441)).
* [Properties](#_Trm00082), which define named characteristics and the actions associated with reading and writing those characteristics ([§10.7](#_Toc00457)).
* Events, which define notifications that can be generated by the class ([§10.8](#_Toc00463)).
* Indexers, which permit [instance](#_Trm00172)s of the class to be indexed in the same way (syntactically) as [array](#_Trm00093)s ([§10.9](#_Toc00468)).
* Operators, which define the expression [operator](#_Trm00090)s that can be applied to [instance](#_Trm00172)s of the class ([§10.10](#_Toc00470)).
* Instance constructors, which implement the actions required to initialize [instance](#_Trm00172)s of the class ([§10.11](#_Toc00474))
* Destructors, which implement the actions to be performed before [instance](#_Trm00172)s of the class are permanently discarded ([§10.13](#_Toc00482)).
* Static constructors, which implement the actions required to initialize the class itself ([§10.12](#_Toc00481)).
* Types, which represent the [types](#_Trm00011) that are local to the class ([§10.3.8](#_Toc00418)).

Members that can contain executable code are collectively known as the *function members* of the class type. The [function members](#_Trm00079) of a class type are the [method](#_Trm00056)s, properties, [event](#_Trm00088)s, [indexer](#_Trm00087)s, [operator](#_Trm00090)s, [instance](#_Trm00172) constructors, [destructor](#_Trm00091)s, and [static constructor](#_Trm00081)s of that class type.

A [class\_declaration](#_Grm00107) creates a new [declaration space](#_Trm00130) ([§3.3](#_Toc00067)), and the [class\_member\_declaration](#_Grm00113)s immediately contained by the [class\_declaration](#_Grm00107) introduce new [members](#_Trm00012) into this [declaration space](#_Trm00130). The following rules apply to [class\_member\_declaration](#_Grm00113)s:

* Instance constructors, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s must have the same name as the immediately enclosing class. All other [members](#_Trm00012) must have names that differ from the name of the immediately enclosing class.
* The name of a constant, field, property, [event](#_Trm00088), or type must differ from the names of all other [members](#_Trm00012) declared in the same class.
* The name of a [method](#_Trm00056) must differ from the names of all other non-[method](#_Trm00056)s declared in the same class. In addition, the [signature](#_Trm00061) ([§3.6](#_Toc00081)) of a [method](#_Trm00056) must differ from the [signature](#_Trm00061)s of all other [method](#_Trm00056)s declared in the same class, and two [method](#_Trm00056)s declared in the same class may not have [signature](#_Trm00061)s that differ solely by ref and out.
* The [signature](#_Trm00061) of an [instance](#_Trm00172) constructor must differ from the [signature](#_Trm00061)s of all other [instance](#_Trm00172) constructors declared in the same class, and two constructors declared in the same class may not have [signature](#_Trm00061)s that differ solely by ref and out.
* The [signature](#_Trm00061) of an [indexer](#_Trm00087) must differ from the [signature](#_Trm00061)s of all other [indexer](#_Trm00087)s declared in the same class.
* The [signature](#_Trm00061) of an [operator](#_Trm00090) must differ from the [signature](#_Trm00061)s of all other [operator](#_Trm00090)s declared in the same class.

The [inherited](#_Trm00136) [members](#_Trm00012) of a class type ([§10.3.3](#_Toc00413)) are not part of the [declaration space](#_Trm00130) of a class. Thus, a derived class is allowed to declare a member with the same name or [signature](#_Trm00061) as an [inherited](#_Trm00136) member (which in effect [hide](#_Trm00132)s the [inherited](#_Trm00136) member).

### The [instance](#_Trm00172) type

Each class declaration has an associated [bound type](#_Trm00182) ([§4.4.3](#_Toc00116)), the ***instance type***. For a [generic class declaration](#_Trm00299), the [instance](#_Trm00172) type is formed by creating a [constructed type](#_Trm00178) ([§4.4](#_Toc00113)) from the type declaration, with each of the supplied type [arguments](#_Trm00062) being the corresponding type parameter. Since the [instance](#_Trm00172) type uses the type [parameters](#_Trm00059), it can only be used where the type [parameters](#_Trm00059) are in [scope](#_Trm00148); that is, inside the class declaration. The [instance](#_Trm00172) type is the type of this for code written inside the class declaration. For non-generic classes, the [instance](#_Trm00172) type is simply the declared class. The following shows several class declarations along with their [instance](#_Trm00172) [types](#_Trm00011):

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 type](#_Trm00178)s

The non-[inherited](#_Trm00136) [members](#_Trm00012) of a [constructed type](#_Trm00178) are obtained by substituting, for each [type\_parameter](#_Grm00032) in the member declaration, the corresponding [type\_argument](#_Grm00031) of the [constructed type](#_Trm00178). The substitution process is based on the semantic meaning of [type declarations](#_Trm00028), and is not simply textual substitution.

For example, given the [generic class declaration](#_Trm00299)

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](#_Trm00178) Gen<int[],IComparable<string>> has the following [members](#_Trm00012):

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](#_Trm00299) Gen is "two-dimensional [array](#_Trm00093) of T", so the type of the member a in the [constructed type](#_Trm00178) above is "two-dimensional [array](#_Trm00093) of one-dimensional [array](#_Trm00093) of int", or int[,][].

Within [instance](#_Trm00172) [function members](#_Trm00079), the type of this is the [instance](#_Trm00172) type ([§10.3.1](#_Toc00411)) of the containing declaration.

All [members](#_Trm00012) of a generic class can use type [parameters](#_Trm00059) from any enclosing class, either directly or as part of a [constructed type](#_Trm00178). When a particular closed [constructed type](#_Trm00178) ([§4.4.2](#_Toc00115)) is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the [constructed type](#_Trm00178). 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](#_Trm00012) of its direct base class type. Inheritance means that a class [implicit](#_Trm00197)ly contains all [members](#_Trm00012) of its direct base class type, except for the [instance](#_Trm00172) constructors, [destructor](#_Trm00091)s and [static constructor](#_Trm00081)s of the base class. Some important aspects of [inheritance](#_Trm00047) are:

* Inheritance is transitive. If C is derived from B, and B is derived from A, then C [inherits](#_Trm00315) the [members](#_Trm00012) declared in B as well as the [members](#_Trm00012) declared in A.
* A derived class extends its direct base class. A derived class can add new [members](#_Trm00012) to those it [inherits](#_Trm00315), but it cannot remove the definition of an [inherited](#_Trm00136) member.
* Instance constructors, [destructor](#_Trm00091)s, and [static constructor](#_Trm00081)s are not [inherited](#_Trm00136), but all other [members](#_Trm00012) are, regardless of their [declared accessibility](#_Trm00140) ([§3.5](#_Toc00076)). However, depending on their [declared accessibility](#_Trm00140), [inherited](#_Trm00136) [members](#_Trm00012) might not be [accessible](#_Trm00138) in a derived class.
* A derived class can ***hide*** ([§3.7.1.2](#_Toc00085)) [inherited](#_Trm00136) [members](#_Trm00012) by declaring new [members](#_Trm00012) with the same name or [signature](#_Trm00061). Note however that hiding an [inherited](#_Trm00136) member does not remove that member—it merely makes that member in[accessible](#_Trm00138) directly through the derived class.
* An [instance](#_Trm00172) of a class contains a set of all [instance](#_Trm00172) fields declared in the class and its [base classes](#_Trm00050), and an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) exists from a derived class type to any of its base [class types](#_Trm00024). Thus, a reference to an [instance](#_Trm00172) of some derived class can be treated as a reference to an [instance](#_Trm00172) of any of its [base classes](#_Trm00050).
* A class can declare virtual [method](#_Trm00056)s, properties, and [indexer](#_Trm00087)s, and [derived classes](#_Trm00049) can override the implementation of these [function members](#_Trm00079). This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the [run-time type](#_Trm00073) of the [instance](#_Trm00172) through which that function member is invoked.

The [inherited](#_Trm00136) member of a constructed class type are the [members](#_Trm00012) of the immediate base class type ([§10.1.4.1](#_Toc00397)), which is found by substituting the type [arguments](#_Trm00062) of the [constructed type](#_Trm00178) for each occurrence of the corresponding type [parameters](#_Trm00059) in the [class\_base](#_Grm00110) specification. These [members](#_Trm00012), in turn, are transformed by substituting, for each [type\_parameter](#_Grm00032) in the member declaration, the corresponding [type\_argument](#_Grm00031) of the [class\_base](#_Grm00110) 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](#_Trm00178) D<int> has a non-[inherited](#_Trm00136) member public int G(string s) obtained by substituting the type argument int for the type parameter T. D<int> also has an [inherited](#_Trm00136) member from the class declaration B. This [inherited](#_Trm00136) 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](#_Trm00136) member public int[] F(long index).

### The new modifier

A [class\_member\_declaration](#_Grm00113) is permitted to declare a member with the same name or [signature](#_Trm00061) as an [inherited](#_Trm00136) member. When this occurs, the derived class member is said to ***hide*** the base class member. Hiding an [inherited](#_Trm00136) 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](#_Trm00132) the base member. This topic is discussed further in [§3.7.1.2](#_Toc00085).

If a new modifier is included in a declaration that doesn't [hide](#_Trm00132) an [inherited](#_Trm00136) member, a warning to that effect is issued. This warning is suppressed by removing the new modifier.

### Access modifiers

A [class\_member\_declaration](#_Grm00113) can have any one of the five possible kinds of [declared accessibility](#_Trm00140) ([§3.5.1](#_Toc00077)): 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](#_Grm00113) does not include any access modifiers, private is assumed.

### Constituent [types](#_Trm00011)

Types that are used in the declaration of a member are called the constituent [types](#_Trm00011) of that member. Possible constituent [types](#_Trm00011) are the type of a constant, field, property, [event](#_Trm00088), or [indexer](#_Trm00087), the [return type](#_Trm00060) of a [method](#_Trm00056) or [operator](#_Trm00090), and the parameter [types](#_Trm00011) of a [method](#_Trm00056), [indexer](#_Trm00087), [operator](#_Trm00090), or [instance](#_Trm00172) constructor. The constituent [types](#_Trm00011) of a member must be at least as [accessible](#_Trm00138) as that member itself ([§3.5.4](#_Toc00080)).

### Static and [instance](#_Trm00172) [members](#_Trm00012)

Members of a class are either ***static members*** or ***instance members***. Generally speaking, it is useful to think of static [members](#_Trm00012) as belonging to [class types](#_Trm00024) and [instance](#_Trm00172) [members](#_Trm00012) as belonging to [object](#_Trm00173)s ([instance](#_Trm00172)s of [class types](#_Trm00024)).

When a field, [method](#_Trm00056), property, [event](#_Trm00088), [operator](#_Trm00090), or constructor declaration includes a static modifier, it declares a static member. In addition, a constant or type declaration [implicit](#_Trm00197)ly declares a static member. Static [members](#_Trm00012) have the following characteristics:

* When a static member M is referenced in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form E.M, E must denote a type containing M. It is a compile-time error for E to denote an [instance](#_Trm00172).
* A [static field](#_Trm00053) identifies exactly one storage location to be shared by all [instance](#_Trm00172)s of a given closed class type. No matter how many [instance](#_Trm00172)s of a given closed class type are created, there is only ever one copy of a [static field](#_Trm00053).
* A static function member ([method](#_Trm00056), property, [event](#_Trm00088), [operator](#_Trm00090), or constructor) does not operate on a specific [instance](#_Trm00172), and it is a compile-time error to refer to this in such a function member.

When a field, [method](#_Trm00056), property, [event](#_Trm00088), [indexer](#_Trm00087), constructor, or [destructor](#_Trm00091) declaration does not include a static modifier, it declares an [instance](#_Trm00172) member. (An [instance](#_Trm00172) member is sometimes called a non-static member.) Instance [members](#_Trm00012) have the following characteristics:

* When an [instance](#_Trm00172) member M is referenced in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form E.M, E must denote an [instance](#_Trm00172) of a type containing M. It is a [binding](#_Trm00210)-time error for E to denote a type.
* Every [instance](#_Trm00172) of a class contains a separate set of all [instance](#_Trm00172) fields of the class.
* An [instance](#_Trm00172) function member ([method](#_Trm00056), property, [indexer](#_Trm00087), [instance](#_Trm00172) constructor, or [destructor](#_Trm00091)) operates on a given [instance](#_Trm00172) of the class, and this [instance](#_Trm00172) can be accessed as this ([§7.6.7](#_Toc00267)).

The following example illustrates the rules for accessing static and [instance](#_Trm00172) [members](#_Trm00012):

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](#_Trm00056) shows that in an [instance](#_Trm00172) function member, a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)) can be used to access both [instance](#_Trm00172) [members](#_Trm00012) and static [members](#_Trm00012). The G [method](#_Trm00056) shows that in a static function member, it is a compile-time error to access an [instance](#_Trm00172) member through a [simple\_name](#_Grm00036). The Main [method](#_Trm00056) shows that in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)), [instance](#_Trm00172) [members](#_Trm00012) must be accessed through [instance](#_Trm00172)s, and static [members](#_Trm00012) must be accessed through [types](#_Trm00011).

### Nested [types](#_Trm00011)

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](#_Trm00143) type because it is declared within class A, and class A is a non-[nested](#_Trm00143) type because it is declared within a compilation unit.

#### Fully qualified name

The [fully qualified name](#_Trm00153) ([§3.8.1](#_Toc00087)) for a [nested](#_Trm00143) type is S.N where S is the [fully qualified name](#_Trm00153) of the type in which type N is declared.

#### Declared accessibility

Non-[nested](#_Trm00143) [types](#_Trm00011) can have public or internal [declared accessibility](#_Trm00140) and have internal [declared accessibility](#_Trm00140) by default. Nested [types](#_Trm00011) can have these forms of [declared accessibility](#_Trm00140) too, plus one or more additional forms of [declared accessibility](#_Trm00140), depending on whether the containing type is a class or struct:

* A [nested](#_Trm00143) type that is declared in a class can have any of five forms of [declared accessibility](#_Trm00140) (public, protected internal, protected, internal, or private) and, like other class [members](#_Trm00012), defaults to private [declared accessibility](#_Trm00140).
* A [nested](#_Trm00143) type that is declared in a struct can have any of three forms of [declared accessibility](#_Trm00140) (public, internal, or private) and, like other struct [members](#_Trm00012), defaults to private [declared accessibility](#_Trm00140).

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](#_Trm00143) class Node.

#### Hiding

A [nested](#_Trm00143) type may [hide](#_Trm00132) ([§3.7.1](#_Toc00083)) a base member. The new modifier is permitted on [nested](#_Trm00143) [type declarations](#_Trm00028) so that hiding can be expressed [explicit](#_Trm00198)ly. 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](#_Trm00143) class M that [hide](#_Trm00132)s the [method](#_Trm00056) M [defined](#_Trm00121) in Base.

#### this access

A [nested](#_Trm00143) type and its containing type do not have a special relationship with regard to [this\_access](#_Grm00041) ([§7.6.7](#_Toc00267)). Specifically, this within a [nested](#_Trm00143) type cannot be used to refer to [instance](#_Trm00172) [members](#_Trm00012) of the containing type. In cases where a [nested](#_Trm00143) type needs access to the [instance](#_Trm00172) [members](#_Trm00012) of its containing type, access can be provided by providing the this for the [instance](#_Trm00172) of the containing type as a constructor argument for the [nested](#_Trm00143) 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](#_Trm00172) of C creates an [instance](#_Trm00172) of Nested and passes its own this to Nested's constructor in order to provide subsequent access to C's [instance](#_Trm00172) [members](#_Trm00012).

#### Access to private and protected [members](#_Trm00012) of the containing type

A [nested](#_Trm00143) type has access to all of the [members](#_Trm00012) that are [accessible](#_Trm00138) to its containing type, including [members](#_Trm00012) of the containing type that have private and protected [declared accessibility](#_Trm00140). 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](#_Trm00143) class Nested. Within Nested, the [method](#_Trm00056) G calls the static [method](#_Trm00056) F [defined](#_Trm00121) in C, and F has private [declared accessibility](#_Trm00140).

A [nested](#_Trm00143) type also may access protected [members](#_Trm00012) [defined](#_Trm00121) 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](#_Trm00143) class Derived.Nested accesses the protected [method](#_Trm00056) F [defined](#_Trm00121) in Derived's base class, Base, by calling through an [instance](#_Trm00172) of Derived.

#### Nested [types](#_Trm00011) in generic classes

A [generic class declaration](#_Trm00299) can contain [nested](#_Trm00143) [type declarations](#_Trm00028). The type [parameters](#_Trm00059) of the enclosing class can be used within the [nested](#_Trm00143) [types](#_Trm00011). A [nested](#_Trm00143) type declaration can contain additional type [parameters](#_Trm00059) that apply only to the [nested](#_Trm00143) type.

Every type declaration contained within a [generic class declaration](#_Trm00299) is [implicit](#_Trm00197)ly a generic type declaration. When writing a reference to a type [nested](#_Trm00143) within a generic type, the containing [constructed type](#_Trm00178), including its type [arguments](#_Trm00062), must be named. However, from within the outer class, the [nested](#_Trm00143) type can be used without qualification; the [instance](#_Trm00172) type of the outer class can be [implicit](#_Trm00197)ly used when constructing the [nested](#_Trm00143) type. The following example shows three different correct ways to refer to a [constructed type](#_Trm00178) 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 [program](#_Trm00109)ming style, a type parameter in a [nested](#_Trm00143) type can [hide](#_Trm00132) 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](#_Trm00088), or [indexer](#_Trm00087), the implementation must reserve two [method](#_Trm00056) [signature](#_Trm00061)s based on the kind of the member declaration, its name, and its type. It is a compile-time error for a [program](#_Trm00109) to declare a member whose [signature](#_Trm00061) matches one of these reserved [signature](#_Trm00061)s, 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](#_Trm00056) [signature](#_Trm00061)s do participate in [inheritance](#_Trm00047) ([§10.3.3](#_Toc00413)), and can be [hidden](#_Trm00150) with the new modifier ([§10.3.4](#_Toc00414)).

The reservation of these names serves three purposes:

* To allow the underlying implementation to use an ordinary identifier as a [method](#_Trm00056) name for get or set access to the C# language feature.
* To allow other languages to interoperate using an ordinary identifier as a [method](#_Trm00056) 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](#_Trm00091) ([§10.13](#_Toc00482)) also causes a [signature](#_Trm00061) to be reserved ([§10.3.9.4](#_Toc00429)).

#### Member names reserved for properties

For a property P ([§10.7](#_Toc00457)) of type T, the following [signature](#_Trm00061)s are reserved:

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

Both [signature](#_Trm00061)s 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](#_Trm00085) P, thus reserving [signature](#_Trm00061)s for get\_P and set\_P [method](#_Trm00056)s. A class B derives from A and [hide](#_Trm00132)s both of these reserved [signature](#_Trm00061)s. The example produces the output:

123  
123  
456

#### Member names reserved for [event](#_Trm00088)s

For an [event](#_Trm00088) E ([§10.8](#_Toc00463)) of [delegate type](#_Trm00107) T, the following [signature](#_Trm00061)s are reserved:

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

#### Member names reserved for [indexer](#_Trm00087)s

For an [indexer](#_Trm00087) ([§10.9](#_Toc00468)) of type T with parameter-list L, the following [signature](#_Trm00061)s are reserved:

T get\_Item(L);  
void set\_Item(L, T value);

Both [signature](#_Trm00061)s are reserved, even if the [indexer](#_Trm00087) is read-only or write-only.

Furthermore the member name Item is reserved.

#### Member names reserved for [destructor](#_Trm00091)s

For a class containing a [destructor](#_Trm00091) ([§10.13](#_Toc00482)), the following [signature](#_Trm00061) is reserved:

void Finalize();

## Constants

A ***constant*** is a class member that represents a [constant](#_Trm00322) [value](#_Trm00209): a [value](#_Trm00209) that can be computed at compile-time. A [constant\_declaration](#_Grm00114) introduces one or more [constant](#_Trm00322)s 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](#_Grm00114) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)), a new modifier ([§10.3.4](#_Toc00414)), and a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)). The [attributes](#_Trm00108) and modifiers apply to all of the [members](#_Trm00012) declared by the [constant\_declaration](#_Grm00114). Even though [constant](#_Trm00322)s are considered static [members](#_Trm00012), a [constant\_declaration](#_Grm00114) neither requires nor allows a static modifier. It is an error for the same modifier to appear multiple times in a [constant](#_Trm00322) declaration.

The [type](#_Grm00028) of a [constant\_declaration](#_Grm00114) specifies the type of the [members](#_Trm00012) introduced by the declaration. The type is followed by a list of [constant\_declarator](#_Grm00077)s, each of which introduces a new member. A [constant\_declarator](#_Grm00077) consists of an [identifier](#_Grm00007) that names the member, followed by an "=" token, followed by a [constant\_expression](#_Grm00068) ([§7.19](#_Toc00346)) that gives the [value](#_Trm00209) of the member.

The [type](#_Grm00028) specified in a [constant](#_Trm00322) declaration must be sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, an [enum\_type](#_Grm00029), or a [reference\_type](#_Grm00030). Each [constant\_expression](#_Grm00068) must yield a [value](#_Trm00209) of the [target](#_Trm00290) type or of a type that can be converted to the [target](#_Trm00290) type by an [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)).

The [type](#_Grm00028) of a [constant](#_Trm00322) must be at least as [accessible](#_Trm00138) as the [constant](#_Trm00322) itself ([§3.5.4](#_Toc00080)).

The [value](#_Trm00209) of a [constant](#_Trm00322) is obtained in an expression using a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)) or a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)).

A [constant](#_Trm00322) can itself participate in a [constant\_expression](#_Grm00068). Thus, a [constant](#_Trm00322) may be used in any construct that requires a [constant\_expression](#_Grm00068). Examples of such con[structs](#_Trm00092) include case labels, goto case [statements](#_Trm00037), enum member declarations, [attributes](#_Trm00108), and other [constant](#_Trm00322) declarations.

As described in [§7.19](#_Toc00346), a [constant\_expression](#_Grm00068) is an expression that can be fully evaluated at compile-time. Since the only way to create a non-[null value](#_Trm00165) of a [reference\_type](#_Grm00030) other than string is to apply the new [operator](#_Trm00090), and since the new [operator](#_Trm00090) is not permitted in a [constant\_expression](#_Grm00068), the only possible [value](#_Trm00209) for [constant](#_Trm00322)s of [reference\_type](#_Grm00030)s other than string is null.

When a symbolic name for a [constant](#_Trm00322) [value](#_Trm00209) is desired, but when the type of that [value](#_Trm00209) is not permitted in a [constant](#_Trm00322) declaration, or when the [value](#_Trm00209) cannot be computed at compile-time by a [constant\_expression](#_Grm00068), a readonly field ([§10.5.2](#_Toc00433)) may be used instead.

A [constant](#_Trm00322) declaration that declares multiple [constant](#_Trm00322)s is equivalent to multiple declarations of single [constant](#_Trm00322)s with the same [attributes](#_Trm00108), 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 [constant](#_Trm00322)s within the same [program](#_Trm00109) as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the [constant](#_Trm00322) 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 [value](#_Trm00209)s 10, 11, and 12. Constant declarations may depend on [constant](#_Trm00322)s from other [program](#_Trm00109)s, but such dependencies are only possible in one direction. Referring to the example above, if A and B were declared in separate [program](#_Trm00109)s, 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](#_Trm00173) or class. A [field\_declaration](#_Grm00115) introduces one or more [field](#_Trm00323)s 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](#_Grm00115) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)), a new modifier ([§10.3.4](#_Toc00414)), a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)), and a static modifier ([§10.5.1](#_Toc00432)). In addition, a [field\_declaration](#_Grm00115) may include a readonly modifier ([§10.5.2](#_Toc00433)) or a volatile modifier ([§10.5.3](#_Toc00436)) but not both. The [attributes](#_Trm00108) and modifiers apply to all of the [members](#_Trm00012) declared by the [field\_declaration](#_Grm00115). It is an error for the same modifier to appear multiple times in a [field](#_Trm00323) declaration.

The [type](#_Grm00028) of a [field\_declaration](#_Grm00115) specifies the type of the [members](#_Trm00012) introduced by the declaration. The type is followed by a list of [variable\_declarator](#_Grm00115)s, each of which introduces a new member. A [variable\_declarator](#_Grm00115) consists of an [identifier](#_Grm00007) that names that member, optionally followed by an "=" token and a [variable\_initializer](#_Grm00115) ([§10.5.5](#_Toc00438)) that gives the initial [value](#_Trm00209) of that member.

The [type](#_Grm00028) of a [field](#_Trm00323) must be at least as [accessible](#_Trm00138) as the [field](#_Trm00323) itself ([§3.5.4](#_Toc00080)).

The [value](#_Trm00209) of a [field](#_Trm00323) is obtained in an expression using a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)) or a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)). The [value](#_Trm00209) of a non-readonly [field](#_Trm00323) is modified using an [assignment](#_Grm00066) ([§7.17](#_Toc00341)). The [value](#_Trm00209) of a non-readonly [field](#_Trm00323) can be both obtained and modified using postfix increment and decrement [operator](#_Trm00090)s ([§7.6.9](#_Toc00269)) and prefix increment and decrement [operator](#_Trm00090)s ([§7.7.5](#_Toc00286)).

A [field](#_Trm00323) declaration that declares multiple [field](#_Trm00323)s is equivalent to multiple declarations of single [field](#_Trm00323)s with the same [attributes](#_Trm00108), 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](#_Trm00172) [field](#_Trm00323)s

When a [field](#_Trm00323) declaration includes a static modifier, the [field](#_Trm00323)s introduced by the declaration are ***static fields***. When no static modifier is present, the [field](#_Trm00323)s introduced by the declaration are ***instance fields***. Static [field](#_Trm00323)s and [instance](#_Trm00172) [field](#_Trm00323)s are two of the several kinds of [variables](#_Trm00031) ([§5](#_Toc00120)) supported by C#, and at times they are referred to as ***static variables*** and ***instance variables***, respectively.

A static [field](#_Trm00323) is not part of a specific [instance](#_Trm00172); instead, it is shared amongst all [instance](#_Trm00172)s of a closed type ([§4.4.2](#_Toc00115)). No matter how many [instance](#_Trm00172)s of a closed class type are created, there is only ever one copy of a static [field](#_Trm00323) for the associated [application](#_Trm00124) 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](#_Trm00172) [field](#_Trm00323) belongs to an [instance](#_Trm00172). Specifically, every [instance](#_Trm00172) of a class contains a separate set of all the [instance](#_Trm00172) [field](#_Trm00323)s of that class.

When a [field](#_Trm00323) is referenced in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form E.M, if M is a static [field](#_Trm00323), E must denote a type containing M, and if M is an [instance](#_Trm00172) [field](#_Trm00323), E must denote an [instance](#_Trm00172) of a type containing M.

The differences between static and [instance](#_Trm00172) [members](#_Trm00012) are discussed further in [§10.3.7](#_Toc00417).

### Readonly [field](#_Trm00323)s

When a [field\_declaration](#_Grm00115) includes a readonly modifier, the [field](#_Trm00323)s introduced by the declaration are ***readonly fields***. Direct assignments to readonly [field](#_Trm00323)s can only occur as part of that declaration or in an [instance](#_Trm00172) constructor or [static constructor](#_Trm00081) in the same class. (A readonly [field](#_Trm00323) can be assigned to multiple times in these contexts.) Specifically, direct assignments to a readonly [field](#_Trm00323) are permitted only in the following contexts:

* In the [variable\_declarator](#_Grm00115) that introduces the [field](#_Trm00323) (by including a [variable\_initializer](#_Grm00115) in the declaration).
* For an [instance](#_Trm00172) [field](#_Trm00323), in the [instance](#_Trm00172) constructors of the class that contains the [field](#_Trm00323) declaration; for a static [field](#_Trm00323), in the [static constructor](#_Trm00081) of the class that contains the [field](#_Trm00323) declaration. These are also the only contexts in which it is valid to pass a readonly [field](#_Trm00323) as an out or ref parameter.

Attempting to assign to a readonly [field](#_Trm00323) or pass it as an out or ref parameter in any other context is a compile-time error.

#### Using static readonly [field](#_Trm00323)s for [constant](#_Trm00322)s

A static readonly [field](#_Trm00323) is useful when a symbolic name for a [constant](#_Trm00322) [value](#_Trm00209) is desired, but when the type of the [value](#_Trm00209) is not permitted in a const declaration, or when the [value](#_Trm00209) 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](#_Trm00012) cannot be declared as const [members](#_Trm00012) because their [value](#_Trm00209)s cannot be computed at compile-time. However, declaring them static readonly instead has much the same effect.

#### Versioning of [constant](#_Trm00322)s and static readonly [field](#_Trm00323)s

Constants and readonly [field](#_Trm00323)s have different binary [versioning](#_Trm00008) semantics. When an expression [references](#_Trm00160) a [constant](#_Trm00322), the [value](#_Trm00209) of the [constant](#_Trm00322) is obtained at compile-time, but when an expression [references](#_Trm00160) a readonly [field](#_Trm00323), the [value](#_Trm00209) of the [field](#_Trm00323) is not obtained until run-time. Consider an [application](#_Trm00124) that consists of two separate [program](#_Trm00109)s:

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](#_Trm00010) denote two [program](#_Trm00109)s that are compiled separately. Because Program1.Utils.X is declared as a static readonly [field](#_Trm00323), the [value](#_Trm00209) output by the Console.WriteLine statement is not known at compile-time, but rather is obtained at run-time. Thus, if the [value](#_Trm00209) of X is changed and Program1 is recompiled, the Console.WriteLine statement will output the new [value](#_Trm00209) even if Program2 isn't recompiled. However, had X been a [constant](#_Trm00322), the [value](#_Trm00209) 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 [field](#_Trm00323)s

When a [field\_declaration](#_Grm00115) includes a volatile modifier, the [field](#_Trm00323)s introduced by that declaration are ***volatile fields***.

For non-volatile [field](#_Trm00323)s, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded [program](#_Trm00109)s that access [field](#_Trm00323)s without synchronization such as that provided by the [lock\_statement](#_Grm00095) ([§8.12](#_Toc00374)). These optimizations can be performed by the compiler, by the run-time system, or by hardware. For volatile [field](#_Trm00323)s, such reordering optimizations are restricted:

* A read of a volatile [field](#_Trm00323) is called a ***volatile read***. A [volatile read](#_Trm00330) has "acquire semantics"; that is, it is guaranteed to occur prior to any [references](#_Trm00160) to memory that occur after it in the instruction [sequence](#_Trm00259).
* A write of a volatile [field](#_Trm00323) is called a ***volatile write***. A [volatile write](#_Trm00331) has "release semantics"; that is, it is guaranteed to happen after any memory [references](#_Trm00160) prior to the write instruction in the instruction [sequence](#_Trm00259).

These restrictions ensure that all threads will observe [volatile write](#_Trm00331)s 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 write](#_Trm00331)s as seen from all threads of execution. The type of a volatile [field](#_Trm00323) must be one of the following:

* A [reference\_type](#_Grm00030).
* The type byte, sbyte, short, ushort, int, uint, char, float, bool, System.IntPtr, orSystem.UIntPtr.
* An [enum\_type](#_Grm00029) 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](#_Trm00056) Main starts a new thread that runs the [method](#_Trm00056) Thread2. This [method](#_Trm00056) stores a [value](#_Trm00209) into a non-volatile [field](#_Trm00323) called result, then stores true in the volatile [field](#_Trm00323) finished. The main thread waits for the [field](#_Trm00323) finished to be set to true, then reads the [field](#_Trm00323) result. Since finished has been declared volatile, the main thread must read the [value](#_Trm00209) 143 from the [field](#_Trm00323) result. If the [field](#_Trm00323) finished had not been declared volatile, then it would be permissible for the store to result to be [visible](#_Trm00152) to the main thread after the store to finished, and hence for the main thread to read the [value](#_Trm00209) 0 from the [field](#_Trm00323) result. Declaring finished as a volatile [field](#_Trm00323) pr[event](#_Trm00088)s any such inconsistency.

### Field initialization

The initial [value](#_Trm00209) of a [field](#_Trm00323), whether it be a static [field](#_Trm00323) or an [instance](#_Trm00172) [field](#_Trm00323), is the [default value](#_Trm00164) ([§5.2](#_Toc00131)) of the [field](#_Trm00323)'s type. It is not possible to observe the [value](#_Trm00209) of a [field](#_Trm00323) before this default initialization has occurred, and a [field](#_Trm00323) 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 value](#_Trm00164)s.

### Variable initializers

Field declarations may include [variable\_initializer](#_Grm00115)s. For static [field](#_Trm00323)s, variable initializers correspond to assignment [statements](#_Trm00037) that are executed during class initialization. For [instance](#_Trm00172) [field](#_Trm00323)s, variable initializers correspond to assignment [statements](#_Trm00037) that are executed when an [instance](#_Trm00172) 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](#_Trm00323) initializers execute and assignments to i and s occur when the [instance](#_Trm00172) [field](#_Trm00323) initializers execute.

The [default value](#_Trm00164) initialization described in [§10.5.4](#_Toc00437) occurs for all [field](#_Trm00323)s, including [field](#_Trm00323)s that have variable initializers. Thus, when a class is initialized, all static [field](#_Trm00323)s in that class are first initialized to their [default value](#_Trm00164)s, and then the static [field](#_Trm00323) initializers are executed in textual order. Likewise, when an [instance](#_Trm00172) of a class is created, all [instance](#_Trm00172) [field](#_Trm00323)s in that [instance](#_Trm00172) are first initialized to their [default value](#_Trm00164)s, and then the [instance](#_Trm00172) [field](#_Trm00323) initializers are executed in textual order.

It is possible for static [field](#_Trm00323)s with variable initializers to be observed in their [default value](#_Trm00164) 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](#_Trm00109) is valid. It results in the output

a = 1, b = 2

because the static [field](#_Trm00323)s a and b are initialized to 0 (the [default value](#_Trm00164) for int) before their initializers are executed. When the initializer for a runs, the [value](#_Trm00209) of b is zero, and so a is initialized to 1. When the initializer for b runs, the [value](#_Trm00209) of a is already 1, and so b is initialized to 2.

#### Static [field](#_Trm00323) initialization

The static [field](#_Trm00323) variable initializers of a class correspond to a [sequence](#_Trm00259) of assignments that are executed in the textual order in which they appear in the class declaration. If a [static constructor](#_Trm00081) ([§10.12](#_Toc00481)) exists in the class, execution of the static [field](#_Trm00323) initializers occurs immediately prior to executing that [static constructor](#_Trm00081). Otherwise, the static [field](#_Trm00323) initializers are executed at an implementation-dependent time prior to the first use of a static [field](#_Trm00323) 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](#_Trm00160) to those [field](#_Trm00323)s. 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 constructor](#_Trm00081)s execute (as [defined](#_Trm00121) in [§10.12](#_Toc00481)) provide that B's [static constructor](#_Trm00081) (and hence B's static [field](#_Trm00323) initializers) must run before A's [static constructor](#_Trm00081) and [field](#_Trm00323) initializers.

#### Instance [field](#_Trm00323) initialization

The [instance](#_Trm00172) [field](#_Trm00323) variable initializers of a class correspond to a [sequence](#_Trm00259) of assignments that are executed immediately upon entry to any one of the [instance](#_Trm00172) constructors ([§10.11.1](#_Toc00475)) of that class. The variable initializers are executed in the textual order in which they appear in the class declaration. The class [instance](#_Trm00172) creation and initialization process is described further in [§10.11](#_Toc00474).

A variable initializer for an [instance](#_Trm00172) [field](#_Trm00323) cannot reference the [instance](#_Trm00172) 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](#_Trm00172) member through a [simple\_name](#_Grm00036). 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](#_Trm00160) a member of the [instance](#_Trm00172) being created.

## Methods

A ***method*** is a member that implements a computation or action that can be performed by an [object](#_Trm00173) or class. Methods are declared using [method\_declaration](#_Grm00116)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'  
 | method\_modifier\_unsafe  
 ;  
  
return\_type:  
 | type  
 | 'void'  
 ;  
  
member\_name:  
 | identifier  
 | interface\_type '.' identifier  
 ;  
  
method\_body:  
 | block  
 | ';'  
 ;

A [method\_declaration](#_Grm00116) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)) and a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)), the new ([§10.3.4](#_Toc00414)), static ([§10.6.2](#_Toc00447)), virtual ([§10.6.3](#_Toc00448)), override ([§10.6.4](#_Toc00449)), sealed ([§10.6.5](#_Toc00450)), abstract ([§10.6.6](#_Toc00451)), and extern ([§10.6.7](#_Toc00452)) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

* The declaration includes a valid combination of access modifiers ([§10.3.5](#_Toc00415)).
* 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](#_Trm00056) that has the async modifier is an async function and follows the rules described in [§10.14](#_Toc00483).

The [return\_type](#_Grm00116) of a [method](#_Trm00056) declaration specifies the type of the [value](#_Trm00209) computed and returned by the [method](#_Trm00056). The [return\_type](#_Grm00116) is void if the [method](#_Trm00056) does not return a [value](#_Trm00209). If the declaration includes the partial modifier, then the [return type](#_Trm00060) must be void.

The [member\_name](#_Grm00116) specifies the name of the [method](#_Trm00056). Unless the [method](#_Trm00056) is an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation ([§13.4.1](#_Toc00544)), the [member\_name](#_Grm00116) is simply an [identifier](#_Grm00007). For an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation, the [member\_name](#_Grm00116) consists of an [interface\_type](#_Grm00030) followed by a "." and an [identifier](#_Grm00007).

The optional [type\_parameter\_list](#_Grm00109) specifies the type [parameters](#_Trm00059) of the [method](#_Trm00056) ([§10.1.3](#_Toc00395)). If a [type\_parameter\_list](#_Grm00109) is specified the [method](#_Trm00056) is a ***generic method***. If the [method](#_Trm00056) has an extern modifier, a [type\_parameter\_list](#_Grm00109) cannot be specified.

The optional [formal\_parameter\_list](#_Grm00117) specifies the [parameters](#_Trm00059) of the [method](#_Trm00056) ([§10.6.1](#_Toc00442)).

The optional [type\_parameter\_constraints\_clause](#_Grm00111)s specify constraints on individual type [parameters](#_Trm00059) ([§10.1.5](#_Toc00399)) and may only be specified if a [type\_parameter\_list](#_Grm00109) is also supplied, and the [method](#_Trm00056) does not have an override modifier.

The [return\_type](#_Grm00116) and each of the [types](#_Trm00011) referenced in the [formal\_parameter\_list](#_Grm00117) of a [method](#_Trm00056) must be at least as [accessible](#_Trm00138) as the [method](#_Trm00056) itself ([§3.5.4](#_Toc00080)).

For abstract and extern [method](#_Trm00056)s, the [method\_body](#_Grm00116) consists simply of a semicolon. For partial [method](#_Trm00056)s the [method\_body](#_Grm00116) may consist of either a semicolon or a [block](#_Grm00071). For all other [method](#_Trm00056)s, the [method\_body](#_Grm00116) consists of a [block](#_Grm00071), which specifies the [statements](#_Trm00037) to execute when the [method](#_Trm00056) is invoked.

If the [method\_body](#_Grm00116) consists of a semicolon, the the declaration may not include the async modifier.

The name, the type parameter list and the formal parameter list of a [method](#_Trm00056) define the [signature](#_Trm00061) ([§3.6](#_Toc00081)) of the [method](#_Trm00056). Specifically, the [signature](#_Trm00061) of a [method](#_Trm00056) consists of its name, the number of type [parameters](#_Trm00059) and the number, modifiers, and [types](#_Trm00011) of its formal [parameters](#_Trm00059). For these purposes, any type parameter of the [method](#_Trm00056) 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](#_Trm00056).The [return type](#_Trm00060) is not part of a [method](#_Trm00056)'s [signature](#_Trm00061), nor are the names of the type [parameters](#_Trm00059) or the formal [parameters](#_Trm00059).

The name of a [method](#_Trm00056) must differ from the names of all other non-[method](#_Trm00056)s declared in the same class. In addition, the [signature](#_Trm00061) of a [method](#_Trm00056) must differ from the [signature](#_Trm00061)s of all other [method](#_Trm00056)s declared in the same class, and two [method](#_Trm00056)s declared in the same class may not have [signature](#_Trm00061)s that differ solely by ref and out.

The [method](#_Trm00056)'s [type\_parameter](#_Grm00032)s are in [scope](#_Trm00148) throughout the [method\_declaration](#_Grm00116), and can be used to form [types](#_Trm00011) throughout that [scope](#_Trm00148) in [return\_type](#_Grm00116), [method\_body](#_Grm00116), and [type\_parameter\_constraints\_clause](#_Grm00111)s but not in [attributes](#_Grm00147).

All formal [parameters](#_Trm00059) and type [parameters](#_Trm00059) must have different names.

### Method [parameters](#_Trm00059)

The [parameters](#_Trm00059) of a [method](#_Trm00056), if any, are declared by the [method](#_Trm00056)'s [formal\_parameter\_list](#_Grm00117).

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](#_Trm00059) of which only the last may be a [parameter\_array](#_Grm00117).

A [fixed\_parameter](#_Grm00117) consists of an optional set of [attributes](#_Grm00147) ([§17](#_Toc00567)), an optional ref, out or this modifier, a [type](#_Grm00028), an [identifier](#_Grm00007) and an optional [default\_argument](#_Grm00117). Each [fixed\_parameter](#_Grm00117) declares a parameter of the given type with the given name. The this modifier designates the [method](#_Trm00056) as an extension [method](#_Trm00056) and is only allowed on the first parameter of a static [method](#_Trm00056). Extension [method](#_Trm00056)s are further described in [§10.6.9](#_Toc00454).

A [fixed\_parameter](#_Grm00117) with a [default\_argument](#_Grm00117) is known as an ***optional parameter***, whereas a [fixed\_parameter](#_Grm00117) without a [default\_argument](#_Grm00117) is a ***required parameter***. A [required parameter](#_Trm00335) may not appear after an [optional parameter](#_Trm00334) in a [formal\_parameter\_list](#_Grm00117).

A ref or out parameter cannot have a [default\_argument](#_Grm00117). The [expression](#_Grm00067) in a [default\_argument](#_Grm00117) must be one of the following:

* a [constant\_expression](#_Grm00068)
* an expression of the form new S() where S is a [value](#_Trm00209) type
* an expression of the form default(S) where S is a [value](#_Trm00209) type

The [expression](#_Grm00067) must be [implicit](#_Trm00197)ly convertible by an identity or nullable [conversion](#_Trm00196) to the type of the parameter.

If [optional parameter](#_Trm00334)s occur in an [implementing partial method declaration](#_Trm00313) ([§10.2.7](#_Toc00408)) , an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation ([§13.4.1](#_Toc00544)) or in a single-parameter [indexer](#_Trm00087) declaration ([§10.9](#_Toc00468)) the compiler should give a warning, since these [members](#_Trm00012) can never be invoked in a way that permits [arguments](#_Trm00062) to be omitted.

A [parameter\_array](#_Grm00117) consists of an optional set of [attributes](#_Grm00147) ([§17](#_Toc00567)), a params modifier, an [array\_type](#_Grm00030), and an [identifier](#_Grm00007). A parameter [array](#_Trm00093) declares a single parameter of the given [array](#_Trm00093) type with the given name. The [array\_type](#_Grm00030) of a parameter [array](#_Trm00093) must be a single-dimensional [array](#_Trm00093) type ([§12.1](#_Toc00519)). In a [method](#_Trm00056) invocation, a parameter [array](#_Trm00093) permits either a single argument of the given [array](#_Trm00093) type to be specified, or it permits zero or more [arguments](#_Trm00062) of the [array](#_Trm00093) [element type](#_Trm00095) to be specified. Parameter [array](#_Trm00093)s are described further in [§10.6.1.4](#_Toc00446).

A [parameter\_array](#_Grm00117) may occur after an [optional parameter](#_Trm00334), but cannot have a [default value](#_Trm00164) -- the omission of [arguments](#_Trm00062) for a [parameter\_array](#_Grm00117) would instead result in the creation of an empty [array](#_Trm00093).

The following example illustrates different kinds of [parameters](#_Trm00059):

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](#_Grm00117) for M, i is a required ref parameter, d is a required [value](#_Trm00209) parameter, b, s, o and t are optional [value](#_Trm00209) [parameters](#_Trm00059) and a is a parameter [array](#_Trm00093).

A [method](#_Trm00056) declaration creates a separate [declaration space](#_Trm00130) for [parameters](#_Trm00059), type [parameters](#_Trm00059) and [local variable](#_Trm00193)s. Names are introduced into this [declaration space](#_Trm00130) by the type parameter list and the formal parameter list of the [method](#_Trm00056) and by [local variable](#_Trm00193) declarations in the [block](#_Grm00071) of the [method](#_Trm00056). It is an error for two [members](#_Trm00012) of a [method](#_Trm00056) [declaration space](#_Trm00130) to have the same name. It is an error for the [method](#_Trm00056) [declaration space](#_Trm00130) and the [local variable](#_Trm00193) [declaration space](#_Trm00130) of a [nested](#_Trm00143) [declaration space](#_Trm00130) to contain [elements](#_Trm00094) with the same name.

A [method](#_Trm00056) invocation ([§7.6.5.1](#_Toc00261)) creates a copy, specific to that invocation, of the formal [parameters](#_Trm00059) and [local variable](#_Trm00193)s of the [method](#_Trm00056), and the argument list of the invocation assigns [value](#_Trm00209)s or variable [references](#_Trm00160) to the newly created formal [parameters](#_Trm00059). Within the [block](#_Grm00071) of a [method](#_Trm00056), formal [parameters](#_Trm00059) can be referenced by their identifiers in [simple\_name](#_Grm00036) expressions ([§7.6.2](#_Toc00254)).

There are four kinds of formal [parameters](#_Trm00059):

* Value [parameters](#_Trm00059), which are declared without any modifiers.
* Reference [parameters](#_Trm00059), which are declared with the ref modifier.
* Output [parameters](#_Trm00059), which are declared with the out modifier.
* Parameter [array](#_Trm00093)s, which are declared with the params modifier.

As described in [§3.6](#_Toc00081), the ref and out modifiers are part of a [method](#_Trm00056)'s [signature](#_Trm00061), but the params modifier is not.

#### Value [parameters](#_Trm00059)

A parameter declared with no modifiers is a [value](#_Trm00209) parameter. A [value](#_Trm00209) parameter corresponds to a [local variable](#_Trm00193) that gets its initial [value](#_Trm00209) from the corresponding argument supplied in the [method](#_Trm00056) invocation.

When a formal parameter is a [value](#_Trm00209) parameter, the corresponding argument in a [method](#_Trm00056) invocation must be an expression that is [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the formal parameter type.

A [method](#_Trm00056) is permitted to assign new [value](#_Trm00209)s to a [value](#_Trm00209) parameter. Such assignments only affect the local storage location represented by the [value](#_Trm00209) parameter—they have no effect on the actual argument given in the [method](#_Trm00056) invocation.

#### Reference [parameters](#_Trm00059)

A parameter declared with a ref modifier is a [reference parameter](#_Trm00064). Unlike a [value](#_Trm00209) parameter, a [reference parameter](#_Trm00064) does not create a new storage location. Instead, a [reference parameter](#_Trm00064) represents the same storage location as the variable given as the argument in the [method](#_Trm00056) invocation.

When a formal parameter is a [reference parameter](#_Trm00064), the corresponding argument in a [method](#_Trm00056) invocation must consist of the [keyword](#_Trm00117) ref followed by a [variable\_reference](#_Grm00033) ([§5.3.3](#_Toc00135)) of the same type as the formal parameter. A variable must be [definitely assigned](#_Trm00068) before it can be passed as a [reference parameter](#_Trm00064).

Within a [method](#_Trm00056), a [reference parameter](#_Trm00064) is always considered [definitely assigned](#_Trm00068).

A [method](#_Trm00056) declared as an iterator ([§10.14](#_Toc00483)) cannot have reference [parameters](#_Trm00059).

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 [value](#_Trm00209)s of i and j.

In a [method](#_Trm00056) that takes reference [parameters](#_Trm00059) 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](#_Trm00172) [field](#_Trm00323) s.

#### Output [parameters](#_Trm00059)

A parameter declared with an out modifier is an [output parameter](#_Trm00065). Similar to a [reference parameter](#_Trm00064), an [output parameter](#_Trm00065) does not create a new storage location. Instead, an [output parameter](#_Trm00065) represents the same storage location as the variable given as the argument in the [method](#_Trm00056) invocation.

When a formal parameter is an [output parameter](#_Trm00065), the corresponding argument in a [method](#_Trm00056) invocation must consist of the [keyword](#_Trm00117) out followed by a [variable\_reference](#_Grm00033) ([§5.3.3](#_Toc00135)) of the same type as the formal parameter. A variable need not be [definitely assigned](#_Trm00068) before it can be passed as an [output parameter](#_Trm00065), but following an invocation where a variable was passed as an [output parameter](#_Trm00065), the variable is considered [definitely assigned](#_Trm00068).

Within a [method](#_Trm00056), just like a [local variable](#_Trm00193), an [output parameter](#_Trm00065) is initially considered unassigned and must be [definitely assigned](#_Trm00068) before its [value](#_Trm00209) is used.

Every [output parameter](#_Trm00065) of a [method](#_Trm00056) must be [definitely assigned](#_Trm00068) before the [method](#_Trm00056) returns.

A [method](#_Trm00056) declared as a partial [method](#_Trm00056) ([§10.2.7](#_Toc00408)) or an iterator ([§10.14](#_Toc00483)) cannot have [output parameter](#_Trm00065)s.

Output [parameters](#_Trm00059) are typically used in [method](#_Trm00056)s that produce multiple return [value](#_Trm00209)s. 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](#_Trm00031) can be unassigned before they are passed to SplitPath, and that they are considered [definitely assigned](#_Trm00068) following the call.

#### Parameter [array](#_Trm00093)s

A parameter declared with a params modifier is a parameter [array](#_Trm00093). If a formal parameter list includes a parameter [array](#_Trm00093), it must be the last parameter in the list and it must be of a single-dimensional [array](#_Trm00093) type. For example, the [types](#_Trm00011) string[] and string[][] can be used as the type of a parameter [array](#_Trm00093), but the type string[,] can not. It is not possible to combine the params modifier with the modifiers ref and out.

A parameter [array](#_Trm00093) permits [arguments](#_Trm00062) to be specified in one of two ways in a [method](#_Trm00056) invocation:

* The argument given for a parameter [array](#_Trm00093) can be a single expression that is [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the parameter [array](#_Trm00093) type. In this case, the parameter [array](#_Trm00093) acts precisely like a [value](#_Trm00209) parameter.
* Alternatively, the invocation can specify zero or more [arguments](#_Trm00062) for the parameter [array](#_Trm00093), where each argument is an expression that is [implicit](#_Trm00197)ly convertible ([§6.1](#_Toc00168)) to the [element type](#_Trm00095) of the parameter [array](#_Trm00093). In this case, the invocation creates an [instance](#_Trm00172) of the parameter [array](#_Trm00093) type with a [length](#_Trm00096) corresponding to the number of [arguments](#_Trm00062), initializes the [elements](#_Trm00094) of the [array](#_Trm00093) [instance](#_Trm00172) with the given argument [value](#_Trm00209)s, and uses the newly created [array](#_Trm00093) [instance](#_Trm00172) as the actual argument.

Except for allowing a variable number of [arguments](#_Trm00062) in an invocation, a parameter [array](#_Trm00093) is precisely equivalent to a [value](#_Trm00209) parameter ([§10.6.1.1](#_Toc00443)) 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](#_Trm00093) a as a [value](#_Trm00209) parameter. The second invocation of F automatically creates a four-element int[] with the given element [value](#_Trm00209)s and passes that [array](#_Trm00093) [instance](#_Trm00172) as a [value](#_Trm00209) parameter. Likewise, the third invocation of F creates a zero-element int[] and passes that [instance](#_Trm00172) as a [value](#_Trm00209) 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](#_Trm00078), a [method](#_Trm00056) with a parameter [array](#_Trm00093) may be applicable either in its [normal form](#_Trm00237) or in its [expanded form](#_Trm00238) ([§7.5.3.1](#_Toc00243)). The [expanded form](#_Trm00238) of a [method](#_Trm00056) is available only if the [normal form](#_Trm00237) of the [method](#_Trm00056) is not applicable and only if an applicable [method](#_Trm00056) with the same [signature](#_Trm00061) as the [expanded form](#_Trm00238) 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 form](#_Trm00238)s of the [method](#_Trm00056) with a parameter [array](#_Trm00093) are already included in the class as regular [method](#_Trm00056)s. These [expanded form](#_Trm00238)s are therefore not considered when performing [overload resolution](#_Trm00078), and the first and third [method](#_Trm00056) invocations thus select the regular [method](#_Trm00056)s. When a class declares a [method](#_Trm00056) with a parameter [array](#_Trm00093), it is not uncommon to also include some of the [expanded form](#_Trm00238)s as regular [method](#_Trm00056)s. By doing so it is possible to avoid the allocation of an [array](#_Trm00093) [instance](#_Trm00172) that occurs when an [expanded form](#_Trm00238) of a [method](#_Trm00056) with a parameter [array](#_Trm00093) is invoked.

When the type of a parameter [array](#_Trm00093) is object[], a potential ambiguity arises between the [normal form](#_Trm00237) of the [method](#_Trm00056) and the expended form for a single object parameter. The reason for the ambiguity is that an object[] is itself [implicit](#_Trm00197)ly 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](#_Trm00237) of F is applicable because an [implicit](#_Trm00197) [conversion](#_Trm00196) exists from the argument type to the parameter type (both are of type object[]). Thus, [overload resolution](#_Trm00078) selects the [normal form](#_Trm00237) of F, and the argument is passed as a regular [value](#_Trm00209) parameter. In the second and third invocations, the [normal form](#_Trm00237) of F is not applicable because no [implicit](#_Trm00197) [conversion](#_Trm00196) exists from the argument type to the parameter type (type object cannot be [implicit](#_Trm00197)ly converted to type object[]). However, the [expanded form](#_Trm00238) of F is applicable, so it is selected by [overload resolution](#_Trm00078). As a result, a one-element object[] is created by the invocation, and the single element of the [array](#_Trm00093) is initialized with the given argument [value](#_Trm00209) (which itself is a reference to an object[]).

### Static and [instance](#_Trm00172) [method](#_Trm00056)s

When a [method](#_Trm00056) declaration includes a static modifier, that [method](#_Trm00056) is said to be a static [method](#_Trm00056). When no static modifier is present, the [method](#_Trm00056) is said to be an [instance](#_Trm00172) [method](#_Trm00056).

A static [method](#_Trm00056) does not operate on a specific [instance](#_Trm00172), and it is a compile-time error to refer to this in a static [method](#_Trm00056).

An [instance](#_Trm00172) [method](#_Trm00056) operates on a given [instance](#_Trm00172) of a class, and that [instance](#_Trm00172) can be accessed as this ([§7.6.7](#_Toc00267)).

When a [method](#_Trm00056) is referenced in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form E.M, if M is a static [method](#_Trm00056), E must denote a type containing M, and if M is an [instance](#_Trm00172) [method](#_Trm00056), E must denote an [instance](#_Trm00172) of a type containing M.

The differences between static and [instance](#_Trm00172) [members](#_Trm00012) are discussed further in [§10.3.7](#_Toc00417).

### Virtual [method](#_Trm00056)s

When an [instance](#_Trm00172) [method](#_Trm00056) declaration includes a virtual modifier, that [method](#_Trm00056) is said to be a virtual [method](#_Trm00056). When no virtual modifier is present, the [method](#_Trm00056) is said to be a non-virtual [method](#_Trm00056).

The implementation of a non-virtual [method](#_Trm00056) is invariant: The implementation is the same whether the [method](#_Trm00056) is invoked on an [instance](#_Trm00172) of the class in which it is declared or an [instance](#_Trm00172) of a derived class. In contrast, the implementation of a virtual [method](#_Trm00056) can be superseded by [derived classes](#_Trm00049). The process of superseding the implementation of an [inherited](#_Trm00136) virtual [method](#_Trm00056) is known as ***overriding*** that [method](#_Trm00056) ([§10.6.4](#_Toc00449)).

In a virtual [method](#_Trm00056) invocation, the ***run-time type*** of the [instance](#_Trm00172) for which that invocation takes place determines the actual [method](#_Trm00056) implementation to invoke. In a non-virtual [method](#_Trm00056) invocation, the ***compile-time type*** of the [instance](#_Trm00172) is the determining factor. In precise terms, when a [method](#_Trm00056) named N is invoked with an argument list A on an [instance](#_Trm00172) with a [compile-time type](#_Trm00074) C and a [run-time type](#_Trm00073) R (where R is either C or a class derived from C), the invocation is processed as follows:

* First, [overload resolution](#_Trm00078) is applied to C, N, and A, to select a specific [method](#_Trm00056) M from the set of [method](#_Trm00056)s declared in and [inherited](#_Trm00136) by C. This is described in [§7.6.5.1](#_Toc00261).
* Then, if M is a non-virtual [method](#_Trm00056), M is invoked.
* Otherwise, M is a virtual [method](#_Trm00056), and the most derived implementation of M with respect to R is invoked.

For every virtual [method](#_Trm00056) declared in or [inherited](#_Trm00136) by a class, there exists a ***most derived implementation*** of the [method](#_Trm00056) with respect to that class. The [most derived implementation](#_Trm00339) of a virtual [method](#_Trm00056) 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](#_Trm00339) of M.
* Otherwise, if R contains an override of M, then this is the [most derived implementation](#_Trm00339) of M.
* Otherwise, the [most derived implementation](#_Trm00339) of M with respect to R is the same as the [most derived implementation](#_Trm00339) of M with respect to the direct base class of R.

The following example illustrates the differences between virtual and non-virtual [method](#_Trm00056)s:

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](#_Trm00056) F and a virtual [method](#_Trm00056) G. The class B introduces a new non-virtual [method](#_Trm00056) F, thus hiding the [inherited](#_Trm00136) F, and also overrides the [inherited](#_Trm00136) [method](#_Trm00056) 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](#_Trm00073) of the [instance](#_Trm00172) (which is B), not the [compile-time type](#_Trm00074) of the [instance](#_Trm00172) (which is A), determines the actual [method](#_Trm00056) implementation to invoke.

Because [method](#_Trm00056)s are allowed to [hide](#_Trm00132) [inherited](#_Trm00136) [method](#_Trm00056)s, it is possible for a class to contain several virtual [method](#_Trm00056)s with the same [signature](#_Trm00061). This does not present an ambiguity problem, since all but the most derived [method](#_Trm00056) are [hidden](#_Trm00150). 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 [method](#_Trm00056)s with the same [signature](#_Trm00061): The one introduced by A and the one introduced by C. The [method](#_Trm00056) introduced by C [hide](#_Trm00132)s the [method](#_Trm00056) [inherited](#_Trm00136) from A. Thus, the override declaration in D overrides the [method](#_Trm00056) introduced by C, and it is not possible for D to override the [method](#_Trm00056) 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](#_Trm00150) virtual [method](#_Trm00056) by accessing an [instance](#_Trm00172) of D through a less derived type in which the [method](#_Trm00056) is not [hidden](#_Trm00150).

### Override [method](#_Trm00056)s

When an [instance](#_Trm00172) [method](#_Trm00056) declaration includes an override modifier, the [method](#_Trm00056) is said to be an ***override method***. An override [method](#_Trm00056) overrides an [inherited](#_Trm00136) virtual [method](#_Trm00056) with the same [signature](#_Trm00061). Whereas a virtual [method](#_Trm00056) declaration introduces a new [method](#_Trm00056), an override [method](#_Trm00056) declaration specializes an existing [inherited](#_Trm00136) virtual [method](#_Trm00056) by providing a new implementation of that [method](#_Trm00056).

The [method](#_Trm00056) [overridden](#_Trm00075) by an override declaration is known as the ***overridden base method***. For an override [method](#_Trm00056) M declared in a class C, the [overridden](#_Trm00075) base [method](#_Trm00056) 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](#_Trm00138) [method](#_Trm00056) is located which has the same [signature](#_Trm00061) as M after substitution of type [arguments](#_Trm00062). For the purposes of locating the [overridden](#_Trm00075) base [method](#_Trm00056), a [method](#_Trm00056) is considered [accessible](#_Trm00138) if it is public, if it is protected, if it is protected internal, or if it is internal and declared in the same [program](#_Trm00109) as C.

A compile-time error occurs unless all of the following are true for an override declaration:

* An [overridden](#_Trm00075) base [method](#_Trm00056) can be located as described above.
* There is exactly one such [overridden](#_Trm00075) base [method](#_Trm00056). This restriction has effect only if the base class type is a [constructed type](#_Trm00178) where the substitution of type [arguments](#_Trm00062) makes the [signature](#_Trm00061) of two [method](#_Trm00056)s the same.
* The [overridden](#_Trm00075) base [method](#_Trm00056) is a virtual, [abstract](#_Trm00076), or override [method](#_Trm00056). In other words, the [overridden](#_Trm00075) base [method](#_Trm00056) cannot be static or non-virtual.
* The [overridden](#_Trm00075) base [method](#_Trm00056) is not a sealed [method](#_Trm00056).
* The override [method](#_Trm00056) and the [overridden](#_Trm00075) base [method](#_Trm00056) have the same [return type](#_Trm00060).
* The override declaration and the [overridden](#_Trm00075) base [method](#_Trm00056) have the same [declared accessibility](#_Trm00140). In other words, an override declaration cannot change the accessibility of the virtual [method](#_Trm00056). However, if the [overridden](#_Trm00075) base [method](#_Trm00056) is protected internal and it is declared in a different assembly than the assembly containing the override [method](#_Trm00056) then the override [method](#_Trm00056)'s [declared accessibility](#_Trm00140) must be protected.
* The override declaration does not specify type-parameter-constraints-clauses. Instead the constraints are [inherited](#_Trm00136) from the [overridden](#_Trm00075) base [method](#_Trm00056). Note that constraints that are type [parameters](#_Trm00059) in the [overridden](#_Trm00075) [method](#_Trm00056) may be replaced by type [arguments](#_Trm00062) in the [inherited](#_Trm00136) constraint. This can lead to constraints that are not legal when [explicit](#_Trm00198)ly specified, such as [value](#_Trm00209) [types](#_Trm00011) or sealed [types](#_Trm00011).

The following example demonstrates how the [overriding](#_Trm00336) 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](#_Trm00075) base [method](#_Trm00056) using a [base\_access](#_Grm00042) ([§7.6.8](#_Toc00268)). 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](#_Trm00056) declared in A. A [base\_access](#_Grm00042) disables the virtual invocation mechanism and simply treats the base [method](#_Trm00056) as a non-virtual [method](#_Trm00056). Had the invocation in B been written ((A)this).PrintFields(), it would recursively invoke the PrintFields [method](#_Trm00056) declared in B, not the one declared in A, since PrintFields is virtual and the [run-time type](#_Trm00073) of ((A)this) is B.

Only by including an override modifier can a [method](#_Trm00056) override another [method](#_Trm00056). In all other cases, a [method](#_Trm00056) with the same [signature](#_Trm00061) as an [inherited](#_Trm00136) [method](#_Trm00056) simply [hide](#_Trm00132)s the [inherited](#_Trm00136) [method](#_Trm00056). In the example

class A  
{  
 public virtual void F() {}  
}  
  
class B: A  
{  
 public virtual void F() {} // Warning, hiding inherited F()  
}

the F [method](#_Trm00056) in B does not include an override modifier and therefore does not override the F [method](#_Trm00056) in A. Rather, the F [method](#_Trm00056) in B [hide](#_Trm00132)s the [method](#_Trm00056) 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](#_Trm00056) in B [hide](#_Trm00132)s the virtual F [method](#_Trm00056) [inherited](#_Trm00136) from A. Since the new F in B has private access, its [scope](#_Trm00148) 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](#_Trm00136) from A.

### Sealed [method](#_Trm00056)s

When an [instance](#_Trm00172) [method](#_Trm00056) declaration includes a sealed modifier, that [method](#_Trm00056) is said to be a ***sealed method***. If an [instance](#_Trm00172) [method](#_Trm00056) declaration includes the sealed modifier, it must also include the override modifier. Use of the sealed modifier pr[event](#_Trm00088)s a derived class from further [overriding](#_Trm00336) the [method](#_Trm00056).

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 [method](#_Trm00056)s: an F [method](#_Trm00056) that has the sealed modifier and a G [method](#_Trm00056) that does not. B's use of the sealed modifier pr[event](#_Trm00088)s C from further [overriding](#_Trm00336) F.

### Abstract [method](#_Trm00056)s

When an [instance](#_Trm00172) [method](#_Trm00056) declaration includes an abstract modifier, that [method](#_Trm00056) is said to be an ***abstract method***. Although an [abstract](#_Trm00076) [method](#_Trm00056) is [implicit](#_Trm00197)ly also a virtual [method](#_Trm00056), it cannot have the modifier virtual.

An [abstract](#_Trm00076) [method](#_Trm00056) declaration introduces a new virtual [method](#_Trm00056) but does not provide an implementation of that [method](#_Trm00056). Instead, non-[abstract](#_Trm00076) [derived classes](#_Trm00049) are required to provide their own implementation by [overriding](#_Trm00336) that [method](#_Trm00056). Because an [abstract](#_Trm00076) [method](#_Trm00056) provides no actual implementation, the [method\_body](#_Grm00116) of an [abstract](#_Trm00076) [method](#_Trm00056) simply consists of a semicolon.

Abstract [method](#_Trm00056) declarations are only permitted in [abstract](#_Trm00076) classes ([§10.1.1.1](#_Toc00391)).

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](#_Trm00076) notion of a geometrical shape [object](#_Trm00173) that can paint itself. The Paint [method](#_Trm00056) is [abstract](#_Trm00076) because there is no meaningful default implementation. The Ellipse and Box classes are concrete Shape implementations. Because these classes are non-[abstract](#_Trm00076), they are required to override the Paint [method](#_Trm00056) and provide an actual implementation.

It is a compile-time error for a [base\_access](#_Grm00042) ([§7.6.8](#_Toc00268)) to reference an [abstract](#_Trm00076) [method](#_Trm00056). 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](#_Trm00160) an [abstract](#_Trm00076) [method](#_Trm00056).

An [abstract](#_Trm00076) [method](#_Trm00056) declaration is permitted to override a virtual [method](#_Trm00056). This allows an [abstract](#_Trm00076) class to force re-implementation of the [method](#_Trm00056) in [derived classes](#_Trm00049), and makes the original implementation of the [method](#_Trm00056) 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](#_Trm00056), class B overrides this [method](#_Trm00056) with an [abstract](#_Trm00076) [method](#_Trm00056), and class C overrides the [abstract](#_Trm00076) [method](#_Trm00056) to provide its own implementation.

### External [method](#_Trm00056)s

When a [method](#_Trm00056) declaration includes an extern modifier, that [method](#_Trm00056) is said to be an ***external method***. External [method](#_Trm00056)s are implemented externally, typically using a language other than C#. Because an [external method](#_Trm00344) declaration provides no actual implementation, the [method\_body](#_Grm00116) of an [external method](#_Trm00344) simply consists of a semicolon. An [external method](#_Trm00344) may not be generic.

The extern modifier is typically used in conjunction with a DllImport attribute ([§17.5.1](#_Toc00587)), allowing [external method](#_Trm00344)s to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of [external method](#_Trm00344)s can be provided.

When an [external method](#_Trm00344) includes a DllImport attribute, the [method](#_Trm00056) 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 [method](#_Trm00056)s (recap)

When a [method](#_Trm00056) declaration includes a partial modifier, that [method](#_Trm00056) is said to be a ***partial method***. Partial [method](#_Trm00056)s can only be declared as [members](#_Trm00012) of partial [types](#_Trm00011) ([§10.2](#_Toc00401)), and are subject to a number of restrictions. Partial [method](#_Trm00056)s are further described in [§10.2.7](#_Toc00408).

### Extension [method](#_Trm00056)s

When the first parameter of a [method](#_Trm00056) includes the this modifier, that [method](#_Trm00056) is said to be an ***extension method***. Extension [method](#_Trm00056)s can only be declared in non-generic, non-[nested](#_Trm00143) [static class](#_Trm00300)es. The first parameter of an [extension method](#_Trm00346) 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](#_Trm00300) that declares two [extension method](#_Trm00346)s:

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](#_Trm00346) is a regular static [method](#_Trm00056). In addition, where its enclosing [static class](#_Trm00300) is in [scope](#_Trm00148), an [extension method](#_Trm00346) can be invoked using [instance](#_Trm00172) [method](#_Trm00056) invocation syntax ([§7.6.5.2](#_Toc00262)), using the receiver expression as the first argument.

The following [program](#_Trm00109) uses the [extension method](#_Trm00346)s 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](#_Trm00056) is available on the string[], and the ToInt32 [method](#_Trm00056) is available on string, because they have been declared as [extension method](#_Trm00346)s. The meaning of the [program](#_Trm00109) is the same as the following, using ordinary static [method](#_Trm00056) 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](#_Grm00116) of a [method](#_Trm00056) declaration consists of either a [block](#_Grm00071) or a semicolon.

Abstract and [external method](#_Trm00344) declarations do not provide a [method](#_Trm00056) implementation, so their [method](#_Trm00056) bodies simply consist of a semicolon. For any other [method](#_Trm00056), the [method](#_Trm00056) body is a [block](#_Trm00038) ([§8.2](#_Toc00350)) that contains the [statements](#_Trm00037) to execute when that [method](#_Trm00056) is invoked.

The ***result type*** of a [method](#_Trm00056) is void if the [return type](#_Trm00060) is void, or if the [method](#_Trm00056) is async and the [return type](#_Trm00060) is System.Threading.Tasks.Task. Otherwise, the [result type](#_Trm00347) of a non-async [method](#_Trm00056) is its [return type](#_Trm00060), and the [result type](#_Trm00347) of an async [method](#_Trm00056) with [return type](#_Trm00060) System.Threading.Tasks.Task<T> is T.

When the [result type](#_Trm00347) of a [method](#_Trm00056) is void, return [statements](#_Trm00037) ([§8.9.4](#_Toc00370)) in that [method](#_Trm00056)'s body are not permitted to specify an expression. If execution of the [method](#_Trm00056) body of a void [method](#_Trm00056) completes normally (that is, control flows off the end of the [method](#_Trm00056) body), that [method](#_Trm00056) simply returns to its current caller.

When the [result type](#_Trm00347) of a [method](#_Trm00056) is not void, each return statement in that [method](#_Trm00056)'s body must specify an expression that is [implicit](#_Trm00197)ly convertible to the [result type](#_Trm00347). The endpoint of the [method](#_Trm00056) body of a [value](#_Trm00209)-returning [method](#_Trm00056) must not be [reachable](#_Trm00265). In other words, in a [value](#_Trm00209)-returning [method](#_Trm00056), control is not permitted to flow off the end of the [method](#_Trm00056) body.

In the example

class A  
{  
 public int F() {} // Error, return value required  
  
 public int G() {  
 return 1;  
 }  
  
 public int H(bool b) {  
 if (b) {  
 return 1;  
 }  
 else {  
 return 0;  
 }  
 }  
}

the [value](#_Trm00209)-returning F [method](#_Trm00056) results in a compile-time error because control can flow off the end of the [method](#_Trm00056) body. The G and H [method](#_Trm00056)s are correct because all possible execution paths end in a return statement that specifies a return [value](#_Trm00209).

### Method [overloading](#_Trm00077)

The [method](#_Trm00056) [overload resolution](#_Trm00078) rules are described in [§7.5.2](#_Toc00227).

## [Properties](#_Trm00082)

A ***property*** is a member that provides access to a characteristic of an [object](#_Trm00173) or a class. Examples of properties include the [length](#_Trm00096) of a string, the size of a font, the caption of a window, the name of a customer, and so on. [Properties](#_Trm00082) are a natural extension of [field](#_Trm00323)s—both are named [members](#_Trm00012) with associated [types](#_Trm00011), and the syntax for accessing [field](#_Trm00323)s and properties is the same. However, unlike [field](#_Trm00323)s, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the [statements](#_Trm00037) to be executed when their [value](#_Trm00209)s are read or written. [Properties](#_Trm00082) thus provide a mechanism for associating actions with the reading and writing of an [object](#_Trm00173)'s [attributes](#_Trm00108); furthermore, they permit such [attributes](#_Trm00108) to be computed.

[Properties](#_Trm00082) are declared using [property\_declaration](#_Grm00118)s:

property\_declaration:  
 | attributes? property\_modifier\* type member\_name '{' accessor\_declarations '}'  
 ;  
  
property\_modifier:  
 | 'new'  
 | 'public'  
 | 'protected'  
 | 'internal'  
 | 'private'  
 | 'static'  
 | 'virtual'  
 | 'sealed'  
 | 'override'  
 | 'abstract'  
 | 'extern'  
 | property\_modifier\_unsafe  
 ;  
  
member\_name:  
 | identifier  
 | interface\_type '.' identifier  
 ;

A [property\_declaration](#_Grm00118) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)) and a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)), the new ([§10.3.4](#_Toc00414)), static ([§10.6.2](#_Toc00447)), virtual ([§10.6.3](#_Toc00448)), override ([§10.6.4](#_Toc00449)), sealed ([§10.6.5](#_Toc00450)), abstract ([§10.6.6](#_Toc00451)), and extern ([§10.6.7](#_Toc00452)) modifiers.

Property declarations are subject to the same rules as [method](#_Trm00056) declarations ([§10.6](#_Toc00441)) with regard to valid combinations of modifiers.

The [type](#_Grm00028) of a [property](#_Trm00348) declaration specifies the type of the [property](#_Trm00348) introduced by the declaration, and the [member\_name](#_Grm00116) specifies the name of the [property](#_Trm00348). Unless the [property](#_Trm00348) is an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation, the [member\_name](#_Grm00116) is simply an [identifier](#_Grm00007). For an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation ([§13.4.1](#_Toc00544)), the [member\_name](#_Grm00116) consists of an [interface\_type](#_Grm00030) followed by a "." and an [identifier](#_Grm00007).

The [type](#_Grm00028) of a [property](#_Trm00348) must be at least as [accessible](#_Trm00138) as the [property](#_Trm00348) itself ([§3.5.4](#_Toc00080)).

The [accessor\_declarations](#_Grm00119), which must be enclosed in "{" and "}" tokens, declare the [accessors](#_Trm00083) ([§10.7.2](#_Toc00459)) of the [property](#_Trm00348). The [accessors](#_Trm00083) specify the executable [statements](#_Trm00037) associated with reading and writing the [property](#_Trm00348).

Even though the syntax for accessing a [property](#_Trm00348) is the same as that for a [field](#_Trm00323), a [property](#_Trm00348) is not classified as a variable. Thus, it is not possible to pass a [property](#_Trm00348) as a ref or out argument.

When a [property](#_Trm00348) declaration includes an extern modifier, the [property](#_Trm00348) is said to be an ***external property***. Because an [external property](#_Trm00350) declaration provides no actual implementation, each of its [accessor\_declarations](#_Grm00119) consists of a semicolon.

### Static and [instance](#_Trm00172) properties

When a [property](#_Trm00348) declaration includes a static modifier, the [property](#_Trm00348) is said to be a ***static property***. When no static modifier is present, the [property](#_Trm00348) is said to be an ***instance property***.

A static [property](#_Trm00348) is not associated with a specific [instance](#_Trm00172), and it is a compile-time error to refer to this in the [accessors](#_Trm00083) of a static [property](#_Trm00348).

An [instance](#_Trm00172) [property](#_Trm00348) is associated with a given [instance](#_Trm00172) of a class, and that [instance](#_Trm00172) can be accessed as this ([§7.6.7](#_Toc00267)) in the [accessors](#_Trm00083) of that [property](#_Trm00348).

When a [property](#_Trm00348) is referenced in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form E.M, if M is a static [property](#_Trm00348), E must denote a type containing M, and if M is an [instance](#_Trm00172) [property](#_Trm00348), E must denote an [instance](#_Trm00172) of a type containing M.

The differences between static and [instance](#_Trm00172) [members](#_Trm00012) are discussed further in [§10.3.7](#_Toc00417).

### Accessors

The [accessor\_declarations](#_Grm00119) of a [property](#_Trm00348) specify the executable [statements](#_Trm00037) associated with reading and writing that [property](#_Trm00348).

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](#_Grm00119), a [set\_accessor\_declaration](#_Grm00119), or both. Each accessor declaration consists of the token get or set followed by an optional [accessor\_modifier](#_Grm00119) and an [accessor\_body](#_Grm00119).

The use of [accessor\_modifier](#_Grm00119)s is governed by the following restrictions:

* An [accessor\_modifier](#_Grm00119) may not be used in an [interface](#_Trm00102) or in an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation.
* For a [property](#_Trm00348) or [indexer](#_Trm00087) that has no override modifer, an [accessor\_modifier](#_Grm00119) is permitted only if the [property](#_Trm00348) or [indexer](#_Trm00087) has both a get and set accessor, and then is permitted only on one of those [accessors](#_Trm00083).
* For a [property](#_Trm00348) or [indexer](#_Trm00087) that includes an override modifer, an accessor must match the [accessor\_modifier](#_Grm00119), if any, of the accessor being [overridden](#_Trm00075).
* The [accessor\_modifier](#_Grm00119) must declare an accessibility that is strictly more restrictive than the [declared accessibility](#_Trm00140) of the [property](#_Trm00348) or [indexer](#_Trm00087) itself. To be precise:
  + If the [property](#_Trm00348) or [indexer](#_Trm00087) has a [declared accessibility](#_Trm00140) of public, the [accessor\_modifier](#_Grm00119) may be either protected internal, internal, protected, or private.
  + If the [property](#_Trm00348) or [indexer](#_Trm00087) has a [declared accessibility](#_Trm00140) of protected internal, the [accessor\_modifier](#_Grm00119) may be either internal, protected, or private.
  + If the [property](#_Trm00348) or [indexer](#_Trm00087) has a [declared accessibility](#_Trm00140) of internal or protected, the [accessor\_modifier](#_Grm00119) must be private.
  + If the [property](#_Trm00348) or [indexer](#_Trm00087) has a [declared accessibility](#_Trm00140) of private, no [accessor\_modifier](#_Grm00119) may be used.

For abstract and extern properties, the [accessor\_body](#_Grm00119) for each accessor specified is simply a semicolon. A non-[abstract](#_Trm00076), non-extern [property](#_Trm00348) may be an ***automatically implemented property***, in which case both get and set [accessors](#_Trm00083) must be given, both with a semicolon body ([§10.7.3](#_Toc00460)). For the [accessors](#_Trm00083) of any other non-[abstract](#_Trm00076), non-extern [property](#_Trm00348), the [accessor\_body](#_Grm00119) is a [block](#_Grm00071) which specifies the [statements](#_Trm00037) to be executed when the corresponding accessor is invoked.

A get accessor corresponds to a parameterless [method](#_Trm00056) with a return [value](#_Trm00209) of the [property](#_Trm00348) type. Except as the [target](#_Trm00290) of an assignment, when a [property](#_Trm00348) is referenced in an expression, the get accessor of the [property](#_Trm00348) is invoked to compute the [value](#_Trm00209) of the [property](#_Trm00348) ([§7.1.1](#_Toc00206)). The body of a get accessor must conform to the rules for [value](#_Trm00209)-returning [method](#_Trm00056)s described in [§10.6.10](#_Toc00455). In particular, all return [statements](#_Trm00037) in the body of a get accessor must specify an expression that is [implicit](#_Trm00197)ly convertible to the [property](#_Trm00348) type. Furthermore, the endpoint of a get accessor must not be [reachable](#_Trm00265).

A set accessor corresponds to a [method](#_Trm00056) with a single [value](#_Trm00209) parameter of the [property](#_Trm00348) type and a void [return type](#_Trm00060). The [implicit](#_Trm00197) parameter of a set accessor is always named value. When a [property](#_Trm00348) is referenced as the [target](#_Trm00290) of an assignment ([§7.17](#_Toc00341)), or as the operand of ++ or -- ([§7.6.9](#_Toc00269), [§7.7.5](#_Toc00286)), the set accessor is invoked with an argument (whose [value](#_Trm00209) is that of the right-hand side of the assignment or the operand of the ++ or -- [operator](#_Trm00090)) that provides the new [value](#_Trm00209) ([§7.17.1](#_Toc00342)). The body of a set accessor must conform to the rules for void [method](#_Trm00056)s described in [§10.6.10](#_Toc00455). In particular, return [statements](#_Trm00037) in the set accessor body are not permitted to specify an expression. Since a set accessor [implicit](#_Trm00197)ly has a parameter named value, it is a compile-time error for a [local variable](#_Trm00193) or [constant](#_Trm00322) declaration in a set accessor to have that name.

Based on the presence or absence of the get and set [accessors](#_Trm00083), a [property](#_Trm00348) is classified as follows:

* A [property](#_Trm00348) that includes both a get accessor and a set accessor is said to be a ***read-write*** [property](#_Trm00348).
* A [property](#_Trm00348) that has only a get accessor is said to be a ***read-only*** [property](#_Trm00348). It is a compile-time error for a [read-only](#_Trm00355) [property](#_Trm00348) to be the [target](#_Trm00290) of an assignment.
* A [property](#_Trm00348) that has only a set accessor is said to be a ***write-only*** [property](#_Trm00348). Except as the [target](#_Trm00290) of an assignment, it is a compile-time error to reference a [write-only](#_Trm00356) [property](#_Trm00348) 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](#_Trm00348). The get accessor of the Caption [property](#_Trm00348) returns the string stored in the private caption [field](#_Trm00323). The set accessor checks if the new [value](#_Trm00209) is different from the current [value](#_Trm00209), and if so, it stores the new [value](#_Trm00209) and repaints the control. [Properties](#_Trm00082) often follow the pattern shown above: The get accessor simply returns a [value](#_Trm00209) stored in a private [field](#_Trm00323), and the set accessor modifies that private [field](#_Trm00323) and then performs any additional actions required to fully update the state of the [object](#_Trm00173).

Given the Button class above, the following is an example of use of the Caption [property](#_Trm00348):

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](#_Trm00209) to the [property](#_Trm00348), and the get accessor is invoked by referencing the [property](#_Trm00348) in an expression.

The get and set [accessors](#_Trm00083) of a [property](#_Trm00348) are not distinct [members](#_Trm00012), and it is not possible to declare the [accessors](#_Trm00083) of a [property](#_Trm00348) separately. As such, it is not possible for the two [accessors](#_Trm00083) of a [read-write](#_Trm00354) [property](#_Trm00348) 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](#_Trm00354) [property](#_Trm00348). Rather, it declares two properties with the same name, one [read-only](#_Trm00355) and one [write-only](#_Trm00356). Since two [members](#_Trm00012) 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](#_Trm00348) by the same name as an [inherited](#_Trm00136) [property](#_Trm00348), the derived [property](#_Trm00348) [hide](#_Trm00132)s the [inherited](#_Trm00136) [property](#_Trm00348) 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](#_Trm00348) in B [hide](#_Trm00132)s the P [property](#_Trm00348) in A with respect to both reading and writing. Thus, in the [statements](#_Trm00037)

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](#_Trm00355) P [property](#_Trm00348) in B [hide](#_Trm00132)s the [write-only](#_Trm00356) P [property](#_Trm00348) in A. Note, however, that a cast can be used to access the [hidden](#_Trm00150) P [property](#_Trm00348).

Unlike public [field](#_Trm00323)s, properties provide a separation between an [object](#_Trm00173)'s internal state and its public [interface](#_Trm00102). 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 [field](#_Trm00323)s, x and y, to store its location. The location is publicly exposed both as an X and a Y [property](#_Trm00348) and as a Location [property](#_Trm00348) 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](#_Trm00102) 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 [field](#_Trm00323)s, 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 [field](#_Trm00323)s directly. In particular, when a [property](#_Trm00348) is non-virtual and contains only a small amount of code, the execution environment may replace calls to [accessors](#_Trm00083) with the actual code of the [accessors](#_Trm00083). This process is known as ***inlining***, and it makes [property](#_Trm00348) access as efficient as [field](#_Trm00323) access, yet preserves the increased flexibility of properties.

Since invoking a get accessor is conceptually equivalent to reading the [value](#_Trm00209) of a [field](#_Trm00323), it is considered bad [program](#_Trm00109)ming style for get [accessors](#_Trm00083) to have observable side-effects. In the example

class Counter  
{  
 private int next;  
  
 public int Next {  
 get { return next++; }  
 }  
}

the [value](#_Trm00209) of the Next [property](#_Trm00348) [depends on](#_Trm00306) the number of times the [property](#_Trm00348) has previously been accessed. Thus, accessing the [property](#_Trm00348) produces an observable side-effect, and the [property](#_Trm00348) should be implemented as a [method](#_Trm00056) instead.

The "no side-effects" convention for get [accessors](#_Trm00083) doesn't mean that get [accessors](#_Trm00083) should always be written to simply return [value](#_Trm00209)s stored in [field](#_Trm00323)s. Indeed, get [accessors](#_Trm00083) often compute the [value](#_Trm00209) of a [property](#_Trm00348) by accessing multiple [field](#_Trm00323)s or invoking [method](#_Trm00056)s. However, a properly designed get accessor performs no actions that cause observable changes in the state of the [object](#_Trm00173).

[Properties](#_Trm00082) can be used to delay initialization of a [resource](#_Trm00296) 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](#_Trm00012) as properties, the Console class can delay their initialization until they are actually used. For example, upon first referencing the Out [property](#_Trm00348), as in

Console.Out.WriteLine("hello, world");

the underlying TextWriter for the output device is created. But if the [application](#_Trm00124) makes no reference to the In and Error properties, then no [object](#_Trm00173)s are created for those devices.

### Automatically implemented properties

When a [property](#_Trm00348) is specified as an [automatically implemented property](#_Trm00353), a [hidden](#_Trm00150) backing [field](#_Trm00323) is automatically available for the [property](#_Trm00348), and the [accessors](#_Trm00083) are implemented to read from and write to that backing [field](#_Trm00323).

The following example:

public class Point {  
 public int X { get; set; } // automatically implemented  
 public int Y { get; set; } // automatically implemented  
}

is equivalent to the following declaration:

public class Point {  
 private int x;  
 private int y;  
 public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Because the backing [field](#_Trm00323) is in[accessible](#_Trm00138), it can be read and written only through the [property](#_Trm00348) [accessors](#_Trm00083), even within the containing type. This means that automatically implemented [read-only](#_Trm00355) or [write-only](#_Trm00356) properties do not make sense, and are disallowed. It is however possible to set the access level of each accessor differently. Thus, the effect of a [read-only](#_Trm00355) [property](#_Trm00348) with a private backing [field](#_Trm00323) can be mimicked like this:

public class ReadOnlyPoint {  
 public int X { get; private set; }  
 public int Y { get; private set; }  
 public ReadOnlyPoint(int x, int y) { X = x; Y = y; }  
}

This restriction also means that definite assignment of [struct types](#_Trm00022) with auto-implemented properties can only be achieved using the standard constructor of the struct, since assigning to the [property](#_Trm00348) itself requires the struct to be [definitely assigned](#_Trm00068). This means that user-[defined](#_Trm00121) constructors must call the [default constructor](#_Trm00163).

### Accessibility

If an accessor has an [accessor\_modifier](#_Grm00119), the [accessibility domain](#_Trm00141) ([§3.5.2](#_Toc00078)) of the accessor is determined using the [declared accessibility](#_Trm00140) of the [accessor\_modifier](#_Grm00119). If an accessor does not have an [accessor\_modifier](#_Grm00119), the [accessibility domain](#_Trm00141) of the accessor is determined from the [declared accessibility](#_Trm00140) of the [property](#_Trm00348) or [indexer](#_Trm00087).

The presence of an [accessor\_modifier](#_Grm00119) never affects member lookup ([§7.3](#_Toc00211)) or [overload resolution](#_Trm00078) ([§7.5.3](#_Toc00242)). The modifiers on the [property](#_Trm00348) or [indexer](#_Trm00087) always determine which [property](#_Trm00348) or [indexer](#_Trm00087) is bound to, regardless of the context of the access.

Once a particular [property](#_Trm00348) or [indexer](#_Trm00087) has been selected, the [accessibility domain](#_Trm00141)s of the specific [accessors](#_Trm00083) involved are used to determine if that usage is valid:

* If the usage is as a [value](#_Trm00209) ([§7.1.1](#_Toc00206)), the get accessor must exist and be [accessible](#_Trm00138).
* If the usage is as the [target](#_Trm00290) of a simple assignment ([§7.17.1](#_Toc00342)), the set accessor must exist and be [accessible](#_Trm00138).
* If the usage is as the [target](#_Trm00290) of compound assignment ([§7.17.2](#_Toc00343)), or as the [target](#_Trm00290) of the ++ or -- [operator](#_Trm00090)s ([§7.5](#_Toc00223).9, [§7.6.5](#_Toc00260)), both the get [accessors](#_Trm00083) and the set accessor must exist and be [accessible](#_Trm00138).

In the following example, the [property](#_Trm00348) A.Text is [hidden](#_Trm00150) by the [property](#_Trm00348) B.Text, even in contexts where only the set accessor is called. In contrast, the [property](#_Trm00348) B.Count is not [accessible](#_Trm00138) to class M, so the [accessible](#_Trm00138) [property](#_Trm00348) 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](#_Trm00102) may not have an [accessor\_modifier](#_Grm00119). If only one accessor is used to implement an [interface](#_Trm00102), the other accessor may be declared with an [accessor\_modifier](#_Grm00119):

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](#_Trm00076) [property](#_Trm00348) [accessors](#_Trm00083)

A virtual [property](#_Trm00348) declaration specifies that the [accessors](#_Trm00083) of the [property](#_Trm00348) are virtual. The virtual modifier applies to both [accessors](#_Trm00083) of a [read-write](#_Trm00354) [property](#_Trm00348)—it is not possible for only one accessor of a [read-write](#_Trm00354) [property](#_Trm00348) to be virtual.

An abstract [property](#_Trm00348) declaration specifies that the [accessors](#_Trm00083) of the [property](#_Trm00348) are virtual, but does not provide an actual implementation of the [accessors](#_Trm00083). Instead, non-[abstract](#_Trm00076) [derived classes](#_Trm00049) are required to provide their own implementation for the [accessors](#_Trm00083) by [overriding](#_Trm00336) the [property](#_Trm00348). Because an accessor for an [abstract](#_Trm00076) [property](#_Trm00348) declaration provides no actual implementation, its [accessor\_body](#_Grm00119) simply consists of a semicolon.

A [property](#_Trm00348) declaration that includes both the abstract and override modifiers specifies that the [property](#_Trm00348) is [abstract](#_Trm00076) and overrides a base [property](#_Trm00348). The [accessors](#_Trm00083) of such a [property](#_Trm00348) are also [abstract](#_Trm00076).

Abstract [property](#_Trm00348) declarations are only permitted in [abstract](#_Trm00076) classes ([§10.1.1.1](#_Toc00391)).The [accessors](#_Trm00083) of an [inherited](#_Trm00136) virtual [property](#_Trm00348) can be [overridden](#_Trm00075) in a derived class by including a [property](#_Trm00348) declaration that specifies an override directive. This is known as an ***overriding property declaration***. An [overriding](#_Trm00336) [property](#_Trm00348) declaration does not declare a new [property](#_Trm00348). Instead, it simply specializes the implementations of the [accessors](#_Trm00083) of an existing virtual [property](#_Trm00348).

An [overriding](#_Trm00336) [property](#_Trm00348) declaration must specify the exact same accessibility modifiers, type, and name as the [inherited](#_Trm00136) [property](#_Trm00348). If the [inherited](#_Trm00136) [property](#_Trm00348) has only a single accessor (i.e., if the [inherited](#_Trm00136) [property](#_Trm00348) is [read-only](#_Trm00355) or [write-only](#_Trm00356)), the [overriding](#_Trm00336) [property](#_Trm00348) must include only that accessor. If the [inherited](#_Trm00136) [property](#_Trm00348) includes both [accessors](#_Trm00083) (i.e., if the [inherited](#_Trm00136) [property](#_Trm00348) is [read-write](#_Trm00354)), the [overriding](#_Trm00336) [property](#_Trm00348) can include either a single accessor or both [accessors](#_Trm00083).

An [overriding](#_Trm00336) [property](#_Trm00348) declaration may include the sealed modifier. Use of this modifier pr[event](#_Trm00088)s a derived class from further [overriding](#_Trm00336) the [property](#_Trm00348). The [accessors](#_Trm00083) of a sealed [property](#_Trm00348) are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and [abstract](#_Trm00076) [accessors](#_Trm00083) behave exactly like virtual, sealed, override and [abstract](#_Trm00076) [method](#_Trm00056)s. Specifically, the rules described in [§10.6.3](#_Toc00448), [§10.6.4](#_Toc00449), [§10.6.5](#_Toc00450), and [§10.6.6](#_Toc00451) apply as if [accessors](#_Trm00083) were [method](#_Trm00056)s of a corresponding form:

* A get accessor corresponds to a parameterless [method](#_Trm00056) with a return [value](#_Trm00209) of the [property](#_Trm00348) type and the same modifiers as the containing [property](#_Trm00348).
* A set accessor corresponds to a [method](#_Trm00056) with a single [value](#_Trm00209) parameter of the [property](#_Trm00348) type, a void [return type](#_Trm00060), and the same modifiers as the containing [property](#_Trm00348).

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](#_Trm00355) [property](#_Trm00348), Y is a virtual [read-write](#_Trm00354) [property](#_Trm00348), and Z is an [abstract](#_Trm00076) [read-write](#_Trm00354) [property](#_Trm00348). Because Z is [abstract](#_Trm00076), the containing class A must also be declared [abstract](#_Trm00076).

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](#_Trm00336) [property](#_Trm00348) declarations. Each [property](#_Trm00348) declaration exactly matches the accessibility modifiers, type, and name of the corresponding [inherited](#_Trm00136) [property](#_Trm00348). The get accessor of X and the set accessor of Y use the base [keyword](#_Trm00117) to access the [inherited](#_Trm00136) [accessors](#_Trm00083). The declaration of Z overrides both [abstract](#_Trm00076) [accessors](#_Trm00083)—thus, there are no outstanding [abstract](#_Trm00076) [function members](#_Trm00079) in B, and B is permitted to be a non-[abstract](#_Trm00076) class.

When a [property](#_Trm00348) is declared as an override, any [overridden](#_Trm00075) [accessors](#_Trm00083) must be [accessible](#_Trm00138) to the [overriding](#_Trm00336) code. In addition, the [declared accessibility](#_Trm00140) of both the [property](#_Trm00348) or [indexer](#_Trm00087) itself, and of the [accessors](#_Trm00083), must match that of the [overridden](#_Trm00075) member and [accessors](#_Trm00083). 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](#_Trm00173) or class to provide notifications. Clients can attach executable code for [event](#_Trm00088)s by supplying ***event handlers***.

Events are declared using [event\_declaration](#_Grm00120)s:

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](#_Grm00120) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)) and a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)), the new ([§10.3.4](#_Toc00414)), static ([§10.6.2](#_Toc00447)), virtual ([§10.6.3](#_Toc00448)), override ([§10.6.4](#_Toc00449)), sealed ([§10.6.5](#_Toc00450)), abstract ([§10.6.6](#_Toc00451)), and extern ([§10.6.7](#_Toc00452)) modifiers.

Event declarations are subject to the same rules as [method](#_Trm00056) declarations ([§10.6](#_Toc00441)) with regard to valid combinations of modifiers.

The [type](#_Grm00028) of an [event](#_Trm00088) declaration must be a [delegate\_type](#_Grm00030) ([§4.2](#_Toc00102)), and that [delegate\_type](#_Grm00030) must be at least as [accessible](#_Trm00138) as the [event](#_Trm00088) itself ([§3.5.4](#_Toc00080)).

An [event](#_Trm00088) declaration may include [event\_accessor\_declarations](#_Grm00120). However, if it does not, for non-extern, non-[abstract](#_Trm00076) [event](#_Trm00088)s, the compiler supplies them automatically ([§10.8.1](#_Toc00464)); for extern [event](#_Trm00088)s, the [accessors](#_Trm00083) are provided externally.

An [event](#_Trm00088) declaration that omits [event\_accessor\_declarations](#_Grm00120) defines one or more [event](#_Trm00088)s—one for each of the [variable\_declarator](#_Grm00115)s. The [attributes](#_Trm00108) and modifiers apply to all of the [members](#_Trm00012) declared by such an [event\_declaration](#_Grm00120).

It is a compile-time error for an [event\_declaration](#_Grm00120) to include both the abstract modifier and brace-delimited [event\_accessor\_declarations](#_Grm00120).

When an [event](#_Trm00088) declaration includes an extern modifier, the [event](#_Trm00088) is said to be an ***external event***. Because an external [event](#_Trm00088) declaration provides no actual implementation, it is an error for it to include both the extern modifier and [event\_accessor\_declarations](#_Grm00120).

It is a compile-time error for a [variable\_declarator](#_Grm00115) of an [event](#_Trm00088) declaration with an abstract or external modifier to include a [variable\_initializer](#_Grm00115).

An [event](#_Trm00088) can be used as the left-hand operand of the += and -= [operator](#_Trm00090)s ([§7.17.3](#_Toc00344)). These [operator](#_Trm00090)s are used, respectively, to attach [event](#_Trm00088) handlers to or to remove [event](#_Trm00088) handlers from an [event](#_Trm00088), and the access modifiers of the [event](#_Trm00088) control the contexts in which such operations are permitted.

Since += and -= are the only operations that are permitted on an [event](#_Trm00088) outside the type that declares the [event](#_Trm00088), external code can add and remove handlers for an [event](#_Trm00088), but cannot in any other way obtain or modify the underlying list of [event](#_Trm00088) handlers.

In an operation of the form x += y or x -= y, when x is an [event](#_Trm00088) 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](#_Trm00209) of x after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an [event](#_Trm00088).

The following example shows how [event](#_Trm00088) handlers are attached to [instance](#_Trm00172)s 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](#_Trm00172) constructor creates two Button [instance](#_Trm00172)s and attaches [event](#_Trm00088) handlers to the Click [event](#_Trm00088)s.

### Field-like [event](#_Trm00088)s

Within the [program](#_Trm00109) text of the class or struct that contains the declaration of an [event](#_Trm00088), certain [event](#_Trm00088)s can be used like [field](#_Trm00323)s. To be used in this way, an [event](#_Trm00088) must not be abstract or extern, and must not [explicit](#_Trm00198)ly include [event\_accessor\_declarations](#_Grm00120). Such an [event](#_Trm00088) can be used in any context that permits a [field](#_Trm00323). The [field](#_Trm00323) contains a delegate ([§15](#_Toc00557)) which refers to the list of [event](#_Trm00088) handlers that have been added to the [event](#_Trm00088). If no [event](#_Trm00088) handlers have been added, the [field](#_Trm00323) 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](#_Trm00323) within the Button class. As the example demonstrates, the [field](#_Trm00323) can be examined, modified, and used in delegate invocation expressions. The OnClick [method](#_Trm00056) in the Button class "raises" the Click [event](#_Trm00088). The notion of raising an [event](#_Trm00088) is precisely equivalent to invoking the delegate represented by the [event](#_Trm00088)—thus, there are no special language con[structs](#_Trm00092) for raising [event](#_Trm00088)s. 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 -= [operator](#_Trm00090)s, as in

b.Click += new EventHandler(...);

which appends a delegate to the invocation list of the Click [event](#_Trm00088), and

b.Click -= new EventHandler(...);

which removes a delegate from the invocation list of the Click [event](#_Trm00088).

When compiling a [field](#_Trm00323)-like [event](#_Trm00088), the compiler automatically creates storage to hold the delegate, and creates [accessors](#_Trm00083) for the [event](#_Trm00088) that add or remove [event](#_Trm00088) handlers to the delegate [field](#_Trm00323). The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock ([§8.12](#_Toc00374)) on the containing [object](#_Trm00173) for an [instance](#_Trm00172) [event](#_Trm00088), or the type [object](#_Trm00173) ([§7.6.10.6](#_Toc00276)) for a static [event](#_Trm00088).

Thus, an [instance](#_Trm00172) [event](#_Trm00088) 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](#_Trm00160) to Ev onthe left-hand side of the+=and-=operators cause the add and remove accessors to be invoked. All other references toEvare compiled to reference the hidden field**Evinstead ([Member access](expressions.md#member-access)). The name "**Ev`" is arbitrary; the [hidden](#_Trm00150) [field](#_Trm00323) could have any name or no name at all.

### Event [accessors](#_Trm00083)

Event declarations typically omit [event\_accessor\_declarations](#_Grm00120), as in the Button example above. One situation for doing so involves the case in which the storage cost of one [field](#_Trm00323) per [event](#_Trm00088) is not acceptable. In such cases, a class can include [event\_accessor\_declarations](#_Grm00120) and use a private mechanism for storing the list of [event](#_Trm00088) handlers.

The [event\_accessor\_declarations](#_Grm00120) of an [event](#_Trm00088) specify the executable [statements](#_Trm00037) associated with adding and removing [event](#_Trm00088) handlers.

The accessor declarations consist of an [add\_accessor\_declaration](#_Grm00120) and a [remove\_accessor\_declaration](#_Grm00120). Each accessor declaration consists of the token add or remove followed by a [block](#_Grm00071). The [block](#_Grm00071) associated with an [add\_accessor\_declaration](#_Grm00120) specifies the [statements](#_Trm00037) to execute when an [event](#_Trm00088) handler is added, and the [block](#_Grm00071) associated with a [remove\_accessor\_declaration](#_Grm00120) specifies the [statements](#_Trm00037) to execute when an [event](#_Trm00088) handler is removed.

Each [add\_accessor\_declaration](#_Grm00120) and [remove\_accessor\_declaration](#_Grm00120) corresponds to a [method](#_Trm00056) with a single [value](#_Trm00209) parameter of the [event](#_Trm00088) type and a void [return type](#_Trm00060). The [implicit](#_Trm00197) parameter of an [event](#_Trm00088) accessor is named value. When an [event](#_Trm00088) is used in an [event](#_Trm00088) assignment, the appropriate [event](#_Trm00088) accessor is used. Specifically, if the assignment [operator](#_Trm00090) is += then the add accessor is used, and if the assignment [operator](#_Trm00090) is -= then the remove accessor is used. In either case, the right-hand operand of the assignment [operator](#_Trm00090) is used as the argument to the [event](#_Trm00088) accessor. The [block](#_Trm00038) of an [add\_accessor\_declaration](#_Grm00120) or a [remove\_accessor\_declaration](#_Grm00120) must conform to the rules for void [method](#_Trm00056)s described in [§10.6.10](#_Toc00455). In particular, return [statements](#_Trm00037) in such a [block](#_Trm00038) are not permitted to specify an expression.

Since an [event](#_Trm00088) accessor [implicit](#_Trm00197)ly has a parameter named value, it is a compile-time error for a [local variable](#_Trm00193) or [constant](#_Trm00322) declared in an [event](#_Trm00088) 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 [event](#_Trm00088)s. The AddEventHandler [method](#_Trm00056) associates a delegate [value](#_Trm00209) with a key, the GetEventHandler [method](#_Trm00056) returns the delegate currently associated with a key, and the RemoveEventHandler [method](#_Trm00056) removes a delegate as an [event](#_Trm00088) handler for the specified [event](#_Trm00088). Presumably, the underlying storage mechanism is designed such that there is no cost for associating a null delegate [value](#_Trm00209) with a key, and thus unhandled [event](#_Trm00088)s consume no storage.

### Static and [instance](#_Trm00172) [event](#_Trm00088)s

When an [event](#_Trm00088) declaration includes a static modifier, the [event](#_Trm00088) is said to be a ***static event***. When no static modifier is present, the [event](#_Trm00088) is said to be an ***instance event***.

A static [event](#_Trm00088) is not associated with a specific [instance](#_Trm00172), and it is a compile-time error to refer to this in the [accessors](#_Trm00083) of a static [event](#_Trm00088).

An [instance](#_Trm00172) [event](#_Trm00088) is associated with a given [instance](#_Trm00172) of a class, and this [instance](#_Trm00172) can be accessed as this ([§7.6.7](#_Toc00267)) in the [accessors](#_Trm00083) of that [event](#_Trm00088).

When an [event](#_Trm00088) is referenced in a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form E.M, if M is a static [event](#_Trm00088), E must denote a type containing M, and if M is an [instance](#_Trm00172) [event](#_Trm00088), E must denote an [instance](#_Trm00172) of a type containing M.

The differences between static and [instance](#_Trm00172) [members](#_Trm00012) are discussed further in [§10.3.7](#_Toc00417).

### Virtual, sealed, override, and [abstract](#_Trm00076) [event](#_Trm00088) [accessors](#_Trm00083)

A virtual [event](#_Trm00088) declaration specifies that the [accessors](#_Trm00083) of that [event](#_Trm00088) are virtual. The virtual modifier applies to both [accessors](#_Trm00083) of an [event](#_Trm00088).

An abstract [event](#_Trm00088) declaration specifies that the [accessors](#_Trm00083) of the [event](#_Trm00088) are virtual, but does not provide an actual implementation of the [accessors](#_Trm00083). Instead, non-[abstract](#_Trm00076) [derived classes](#_Trm00049) are required to provide their own implementation for the [accessors](#_Trm00083) by [overriding](#_Trm00336) the [event](#_Trm00088). Because an [abstract](#_Trm00076) [event](#_Trm00088) declaration provides no actual implementation, it cannot provide brace-delimited [event\_accessor\_declarations](#_Grm00120).

An [event](#_Trm00088) declaration that includes both the abstract and override modifiers specifies that the [event](#_Trm00088) is [abstract](#_Trm00076) and overrides a base [event](#_Trm00088). The [accessors](#_Trm00083) of such an [event](#_Trm00088) are also [abstract](#_Trm00076).

Abstract [event](#_Trm00088) declarations are only permitted in [abstract](#_Trm00076) classes ([§10.1.1.1](#_Toc00391)).

The [accessors](#_Trm00083) of an [inherited](#_Trm00136) virtual [event](#_Trm00088) can be [overridden](#_Trm00075) in a derived class by including an [event](#_Trm00088) declaration that specifies an override modifier. This is known as an ***overriding event declaration***. An [overriding](#_Trm00336) [event](#_Trm00088) declaration does not declare a new [event](#_Trm00088). Instead, it simply specializes the implementations of the [accessors](#_Trm00083) of an existing virtual [event](#_Trm00088).

An [overriding](#_Trm00336) [event](#_Trm00088) declaration must specify the exact same accessibility modifiers, type, and name as the [overridden](#_Trm00075) [event](#_Trm00088).

An [overriding](#_Trm00336) [event](#_Trm00088) declaration may include the sealed modifier. Use of this modifier pr[event](#_Trm00088)s a derived class from further [overriding](#_Trm00336) the [event](#_Trm00088). The [accessors](#_Trm00083) of a sealed [event](#_Trm00088) are also sealed.

It is a compile-time error for an [overriding](#_Trm00336) [event](#_Trm00088) declaration to include a new modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and [abstract](#_Trm00076) [accessors](#_Trm00083) behave exactly like virtual, sealed, override and [abstract](#_Trm00076) [method](#_Trm00056)s. Specifically, the rules described in [§10.6.3](#_Toc00448), [§10.6.4](#_Toc00449), [§10.6.5](#_Toc00450), and [§10.6.6](#_Toc00451) apply as if [accessors](#_Trm00083) were [method](#_Trm00056)s of a corresponding form. Each accessor corresponds to a [method](#_Trm00056) with a single [value](#_Trm00209) parameter of the [event](#_Trm00088) type, a void [return type](#_Trm00060), and the same modifiers as the containing [event](#_Trm00088).

## Indexers

An ***indexer*** is a member that enables an [object](#_Trm00173) to be indexed in the same way as an [array](#_Trm00093). Indexers are declared using [indexer\_declaration](#_Grm00121)s:

indexer\_declaration:  
 | attributes? indexer\_modifier\* indexer\_declarator '{' accessor\_declarations '}'  
 ;  
  
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 ']'  
 ;

An [indexer\_declaration](#_Grm00121) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)) and a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)), the new ([§10.3.4](#_Toc00414)), virtual ([§10.6.3](#_Toc00448)), override ([§10.6.4](#_Toc00449)), sealed ([§10.6.5](#_Toc00450)), abstract ([§10.6.6](#_Toc00451)), and extern ([§10.6.7](#_Toc00452)) modifiers.

Indexer declarations are subject to the same rules as [method](#_Trm00056) declarations ([§10.6](#_Toc00441)) with regard to valid combinations of modifiers, with the one exception being that the static modifier is not permitted on an [indexer](#_Trm00087) 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](#_Trm00076) [indexer](#_Trm00087) can override a virtual one.

The [type](#_Grm00028) of an [indexer](#_Trm00087) declaration specifies the [element type](#_Trm00095) of the [indexer](#_Trm00087) introduced by the declaration. Unless the [indexer](#_Trm00087) is an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation, the [type](#_Grm00028) is followed by the [keyword](#_Trm00117) this. For an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation, the [type](#_Grm00028) is followed by an [interface\_type](#_Grm00030), a ".", and the [keyword](#_Trm00117) this. Unlike other [members](#_Trm00012), [indexer](#_Trm00087)s do not have user-[defined](#_Trm00121) names.

The [formal\_parameter\_list](#_Grm00117) specifies the [parameters](#_Trm00059) of the [indexer](#_Trm00087). The formal parameter list of an [indexer](#_Trm00087) corresponds to that of a [method](#_Trm00056) ([§10.6.1](#_Toc00442)), except that at least one parameter must be specified, and that the ref and out parameter modifiers are not permitted.

The [type](#_Grm00028) of an [indexer](#_Trm00087) and each of the [types](#_Trm00011) referenced in the [formal\_parameter\_list](#_Grm00117) must be at least as [accessible](#_Trm00138) as the [indexer](#_Trm00087) itself ([§3.5.4](#_Toc00080)).

The [accessor\_declarations](#_Grm00119) ([§10.7.2](#_Toc00459)), which must be enclosed in "{" and "}" tokens, declare the [accessors](#_Trm00083) of the [indexer](#_Trm00087). The [accessors](#_Trm00083) specify the executable [statements](#_Trm00037) associated with reading and writing [indexer](#_Trm00087) [elements](#_Trm00094).

Even though the syntax for accessing an [indexer](#_Trm00087) element is the same as that for an [array](#_Trm00093) element, an [indexer](#_Trm00087) element is not classified as a variable. Thus, it is not possible to pass an [indexer](#_Trm00087) element as a ref or out argument.

The formal parameter list of an [indexer](#_Trm00087) defines the [signature](#_Trm00061) ([§3.6](#_Toc00081)) of the [indexer](#_Trm00087). Specifically, the [signature](#_Trm00061) of an [indexer](#_Trm00087) consists of the number and [types](#_Trm00011) of its formal [parameters](#_Trm00059). The [element type](#_Trm00095) and names of the formal [parameters](#_Trm00059) are not part of an [indexer](#_Trm00087)'s [signature](#_Trm00061).

The [signature](#_Trm00061) of an [indexer](#_Trm00087) must differ from the [signature](#_Trm00061)s of all other [indexer](#_Trm00087)s declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

* A [property](#_Trm00348) is identified by its name, whereas an [indexer](#_Trm00087) is identified by its [signature](#_Trm00061).
* A [property](#_Trm00348) is accessed through a [simple\_name](#_Grm00036) ([§7.6.2](#_Toc00254)) or a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)), whereas an [indexer](#_Trm00087) element is accessed through an [element\_access](#_Grm00040) ([§7.6.6.2](#_Toc00266)).
* A [property](#_Trm00348) can be a static member, whereas an [indexer](#_Trm00087) is always an [instance](#_Trm00172) member.
* A get accessor of a [property](#_Trm00348) corresponds to a [method](#_Trm00056) with no [parameters](#_Trm00059), whereas a get accessor of an [indexer](#_Trm00087) corresponds to a [method](#_Trm00056) with the same formal parameter list as the [indexer](#_Trm00087).
* A set accessor of a [property](#_Trm00348) corresponds to a [method](#_Trm00056) with a single parameter named value, whereas a set accessor of an [indexer](#_Trm00087) corresponds to a [method](#_Trm00056) with the same formal parameter list as the [indexer](#_Trm00087), plus an additional parameter named value.
* It is a compile-time error for an [indexer](#_Trm00087) accessor to declare a [local variable](#_Trm00193) with the same name as an [indexer](#_Trm00087) parameter.
* In an [overriding](#_Trm00336) [property](#_Trm00348) declaration, the [inherited](#_Trm00136) [property](#_Trm00348) is accessed using the syntax base.P, where P is the [property](#_Trm00348) name. In an [overriding](#_Trm00336) [indexer](#_Trm00087) declaration, the [inherited](#_Trm00136) [indexer](#_Trm00087) is accessed using the syntax base[E], where E is a comma separated list of expressions.

Aside from these differences, all rules [defined](#_Trm00121) in [§10.7.2](#_Toc00459) and [§10.7.3](#_Toc00460) apply to [indexer](#_Trm00087) [accessors](#_Trm00083) as well as to [property](#_Trm00348) [accessors](#_Trm00083).

When an [indexer](#_Trm00087) declaration includes an extern modifier, the [indexer](#_Trm00087) is said to be an ***external indexer***. Because an [external indexer](#_Trm00366) declaration provides no actual implementation, each of its [accessor\_declarations](#_Grm00119) consists of a semicolon.

The example below declares a BitArray class that implements an [indexer](#_Trm00087) for accessing the individual bits in the bit [array](#_Trm00093).

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](#_Trm00172) of the BitArray class consumes substantially less memory than a corresponding bool[] (since each [value](#_Trm00209) 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](#_Trm00094) of the BitArray is precisely the same as for a bool[].

The following example shows a 26 \* 10 grid class that has an [indexer](#_Trm00087) with two [parameters](#_Trm00059). 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](#_Trm00077)

The [indexer](#_Trm00087) [overload resolution](#_Trm00078) rules are described in [§7.5.2](#_Toc00227).

## Operators

An ***operator*** is a member that defines the meaning of an expression [operator](#_Trm00090) that can be applied to [instance](#_Trm00172)s of the class. Operators are declared using [operator\_declaration](#_Grm00122)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  
 | ';'  
 ;

There are three categories of overloadable [operator](#_Trm00090)s: Unary [operator](#_Trm00090)s ([§10.10.1](#_Toc00471)), binary [operator](#_Trm00090)s ([§10.10.2](#_Toc00472)), and [conversion](#_Trm00196) [operator](#_Trm00090)s ([§10.10.3](#_Toc00473)).

When an [operator](#_Trm00090) declaration includes an extern modifier, the [operator](#_Trm00090) is said to be an ***external operator***. Because an [external operator](#_Trm00368) provides no actual implementation, its [operator\_body](#_Grm00122) consists of a semi-colon. For all other [operator](#_Trm00090)s, the [operator\_body](#_Grm00122) consists of a [block](#_Grm00071), which specifies the [statements](#_Trm00037) to execute when the [operator](#_Trm00090) is invoked. The [block](#_Grm00071) of an [operator](#_Trm00090) must conform to the rules for [value](#_Trm00209)-returning [method](#_Trm00056)s described in [§10.6.10](#_Toc00455).

The following rules apply to all [operator](#_Trm00090) declarations:

* An [operator](#_Trm00090) declaration must include both a public and a static modifier.
* The parameter(s) of an [operator](#_Trm00090) must be [value](#_Trm00209) [parameters](#_Trm00059) ([§5.1.4](#_Toc00127)). It is a compile-time error for an [operator](#_Trm00090) declaration to specify ref or out [parameters](#_Trm00059).
* The [signature](#_Trm00061) of an [operator](#_Trm00090) ([§10.10.1](#_Toc00471), [§10.10.2](#_Toc00472), [§10.10.3](#_Toc00473)) must differ from the [signature](#_Trm00061)s of all other [operator](#_Trm00090)s declared in the same class.
* All [types](#_Trm00011) referenced in an [operator](#_Trm00090) declaration must be at least as [accessible](#_Trm00138) as the [operator](#_Trm00090) itself ([§3.5.4](#_Toc00080)).
* It is an error for the same modifier to appear multiple times in an [operator](#_Trm00090) declaration.

Each [operator](#_Trm00090) category imposes additional restrictions, as described in the following sections.

Like other [members](#_Trm00012), [operator](#_Trm00090)s declared in a base class are [inherited](#_Trm00136) by [derived classes](#_Trm00049). Because [operator](#_Trm00090) declarations always require the class or struct in which the [operator](#_Trm00090) is declared to participate in the [signature](#_Trm00061) of the [operator](#_Trm00090), it is not possible for an [operator](#_Trm00090) declared in a derived class to [hide](#_Trm00132) an [operator](#_Trm00090) declared in a base class. Thus, the new modifier is never required, and therefore never permitted, in an [operator](#_Trm00090) declaration.

Additional information on unary and binary [operator](#_Trm00090)s can be found in [§7.3](#_Toc00211).

Additional information on [conversion](#_Trm00196) [operator](#_Trm00090)s can be found in [§6.4](#_Toc00193).

### Unary [operator](#_Trm00090)s

The following rules apply to unary [operator](#_Trm00090) declarations, where T denotes the [instance](#_Trm00172) type of the class or struct that contains the [operator](#_Trm00090) declaration:

* A unary +, -, !, or ~ [operator](#_Trm00090) must take a single parameter of type T or T? and can return any type.
* A unary ++ or -- [operator](#_Trm00090) 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](#_Trm00090) must take a single parameter of type T or T? and must [return type](#_Trm00060) bool.

The [signature](#_Trm00061) of a unary [operator](#_Trm00090) consists of the [operator](#_Trm00090) token (+, -, !, ~, ++, --, true, or false) and the type of the single formal parameter. The [return type](#_Trm00060) is not part of a unary [operator](#_Trm00090)'s [signature](#_Trm00061), nor is the name of the formal parameter.

The true and false unary [operator](#_Trm00090)s require pair-wise declaration. A compile-time error occurs if a class declares one of these [operator](#_Trm00090)s without also declaring the other. The true and false [operator](#_Trm00090)s are described further in [§7.12.2](#_Toc00318) and [§7.20](#_Toc00347).

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](#_Trm00090) [method](#_Trm00056) returns the [value](#_Trm00209) produced by adding 1 to the operand, just like the postfix increment and decrement [operator](#_Trm00090)s ([§7.6.9](#_Toc00269)), and the prefix increment and decrement [operator](#_Trm00090)s ([§7.7.5](#_Toc00286)). Unlike in C++, this [method](#_Trm00056) need not modify the [value](#_Trm00209) of its operand directly. In fact, modifying the operand [value](#_Trm00209) would violate the standard semantics of the postfix increment [operator](#_Trm00090).

### Binary [operator](#_Trm00090)s

The following rules apply to binary [operator](#_Trm00090) declarations, where T denotes the [instance](#_Trm00172) type of the class or struct that contains the [operator](#_Trm00090) declaration:

* A binary non-shift [operator](#_Trm00090) must take two [parameters](#_Trm00059), at least one of which must have type T or T?, and can return any type.
* A binary << or >> [operator](#_Trm00090) must take two [parameters](#_Trm00059), 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](#_Trm00061) of a binary [operator](#_Trm00090) consists of the [operator](#_Trm00090) token (+, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and the [types](#_Trm00011) of the two formal [parameters](#_Trm00059). The [return type](#_Trm00060) and the names of the formal [parameters](#_Trm00059) are not part of a binary [operator](#_Trm00090)'s [signature](#_Trm00061).

Certain binary [operator](#_Trm00090)s require pair-wise declaration. For every declaration of either [operator](#_Trm00090) of a pair, there must be a matching declaration of the other [operator](#_Trm00090) of the pair. Two [operator](#_Trm00090) declarations match when they have the same [return type](#_Trm00060) and the same type for each parameter. The following [operator](#_Trm00090)s require pair-wise declaration:

* operator == and operator !=
* operator > and operator <
* operator >= and operator <=

### Conversion [operator](#_Trm00090)s

A [conversion](#_Trm00196) [operator](#_Trm00090) declaration introduces a ***user-defined conversion*** ([§6.4](#_Toc00193)) which augments the pre-[defined](#_Trm00121) [implicit](#_Trm00197) and [explicit](#_Trm00198) [conversion](#_Trm00196)s.

A [conversion](#_Trm00196) [operator](#_Trm00090) declaration that includes the implicit [keyword](#_Trm00117) introduces a user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196). Implicit [conversion](#_Trm00196)s can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in [§6.1](#_Toc00168).

A [conversion](#_Trm00196) [operator](#_Trm00090) declaration that includes the explicit [keyword](#_Trm00117) introduces a user-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196). Explicit [conversion](#_Trm00196)s can occur in cast expressions, and are described further in [§6.2](#_Toc00181).

A [conversion](#_Trm00196) [operator](#_Trm00090) converts from a [source type](#_Trm00202), indicated by the parameter type of the [conversion](#_Trm00196) [operator](#_Trm00090), to a [target](#_Trm00290) type, indicated by the [return type](#_Trm00060) of the [conversion](#_Trm00196) [operator](#_Trm00090).

For a given [source type](#_Trm00202) S and [target](#_Trm00290) type T, if S or T are [nullable types](#_Trm00023), let S0 and T0 refer to their underlying [types](#_Trm00011), otherwise S0 and T0 are equal to S and T respectively. A class or struct is permitted to declare a [conversion](#_Trm00196) from a [source type](#_Trm00202) S to a [target](#_Trm00290) type T only if all of the following are true:

* S0 and T0 are different [types](#_Trm00011).
* Either S0 or T0 is the class or struct type in which the [operator](#_Trm00090) declaration takes place.
* Neither S0 nor T0 is an [interface\_type](#_Grm00030).
* Excluding user-[defined](#_Trm00121) [conversion](#_Trm00196)s, a [conversion](#_Trm00196) does not exist from S to T or from T to S.

For the purposes of these rules, any type [parameters](#_Trm00059) associated with S or T are considered to be unique [types](#_Trm00011) that have no [inheritance](#_Trm00047) relationship with other [types](#_Trm00011), and any constraints on those type [parameters](#_Trm00059) 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](#_Trm00090) declarations are permitted because, for the purposes of [§10.9](#_Toc00468).3, T and int and string respectively are considered unique [types](#_Trm00011) with no relationship. However, the third [operator](#_Trm00090) is an error because C<T> is the base class of D<T>.

From the second rule it follows that a [conversion](#_Trm00196) [operator](#_Trm00090) must convert either to or from the class or struct type in which the [operator](#_Trm00090) is declared. For example, it is possible for a class or struct type C to define a [conversion](#_Trm00196) 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](#_Trm00121) [conversion](#_Trm00196). Thus, [conversion](#_Trm00196) [operator](#_Trm00090)s are not allowed to convert from or to object because [implicit](#_Trm00197) and [explicit](#_Trm00198) [conversion](#_Trm00196)s already exist between object and all other [types](#_Trm00011). Likewise, neither the source nor the [target](#_Trm00290) [types](#_Trm00011) of a [conversion](#_Trm00196) can be a base type of the other, since a [conversion](#_Trm00196) would then already exist.

However, it is possible to declare [operator](#_Trm00090)s on [generic types](#_Trm00158) that, for particular type [arguments](#_Trm00062), specify [conversion](#_Trm00196)s that already exist as pre-[defined](#_Trm00121) [conversion](#_Trm00196)s. 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](#_Trm00090) declares a [conversion](#_Trm00196) that already exists (an [implicit](#_Trm00197), and therefore also an [explicit](#_Trm00198), [conversion](#_Trm00196) exists from any type to type object).

In cases where a pre-[defined](#_Trm00121) [conversion](#_Trm00196) exists between two [types](#_Trm00011), any user-[defined](#_Trm00121) [conversion](#_Trm00196)s between those [types](#_Trm00011) are ignored. Specifically:

* If a pre-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.1](#_Toc00168)) exists from type S to type T, all user-[defined](#_Trm00121) [conversion](#_Trm00196)s ([implicit](#_Trm00197) or [explicit](#_Trm00198)) from S to T are ignored.
* If a pre-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196) ([§6.2](#_Toc00181)) exists from type S to type T, any user-[defined](#_Trm00121) [explicit](#_Trm00198) [conversion](#_Trm00196)s from S to T are ignored. Furthermore:

If T is an [interface](#_Trm00102) type, user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s from S to T are ignored.

Otherwise, user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s from S to T are still considered.

For all [types](#_Trm00011) but object, the [operator](#_Trm00090)s declared by the Convertible<T> type above do not conflict with pre-[defined](#_Trm00121) [conversion](#_Trm00196)s. 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](#_Trm00121) [conversion](#_Trm00196)s [hide](#_Trm00132) the user-[defined](#_Trm00121) [conversion](#_Trm00196)s 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](#_Trm00121) [conversion](#_Trm00196)s are not allowed to convert from or to [interface\_type](#_Grm00030)s. In particular, this restriction ensures that no user-[defined](#_Trm00121) transformations occur when converting to an [interface\_type](#_Grm00030), and that a [conversion](#_Trm00196) to an [interface\_type](#_Grm00030) succeeds only if the [object](#_Trm00173) being converted actually implements the specified [interface\_type](#_Grm00030).

The [signature](#_Trm00061) of a [conversion](#_Trm00196) [operator](#_Trm00090) consists of the [source type](#_Trm00202) and the [target](#_Trm00290) type. (Note that this is the only form of member for which the [return type](#_Trm00060) participates in the [signature](#_Trm00061).) The implicit or explicit classification of a [conversion](#_Trm00196) [operator](#_Trm00090) is not part of the [operator](#_Trm00090)'s [signature](#_Trm00061). Thus, a class or struct cannot declare both an implicit and an explicit [conversion](#_Trm00196) [operator](#_Trm00090) with the same source and [target](#_Trm00290) [types](#_Trm00011).

In general, user-[defined](#_Trm00121) [implicit](#_Trm00197) [conversion](#_Trm00196)s should be designed to never throw exceptions and never lose information. If a user-[defined](#_Trm00121) [conversion](#_Trm00196) 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](#_Trm00196) should be [defined](#_Trm00121) as an [explicit](#_Trm00198) [conversion](#_Trm00196).

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](#_Trm00196) from Digit to byte is [implicit](#_Trm00197) because it never throws exceptions or loses information, but the [conversion](#_Trm00196) from byte to Digit is [explicit](#_Trm00198) since Digit can only represent a subset of the possible [value](#_Trm00209)s of a byte.

## Instance constructors

An ***instance constructor*** is a member that implements the actions required to initialize an [instance](#_Trm00172) of a class. Instance constructors are declared using [constructor\_declaration](#_Grm00123)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](#_Grm00123) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)), a valid combination of the four access modifiers ([§10.3.5](#_Toc00415)), and an extern ([§10.6.7](#_Toc00452)) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The [identifier](#_Grm00007) of a [constructor\_declarator](#_Grm00123) must name the class in which the [instance](#_Trm00172) constructor is declared. If any other name is specified, a compile-time error occurs.

The optional [formal\_parameter\_list](#_Grm00117) of an [instance](#_Trm00172) constructor is subject to the same rules as the [formal\_parameter\_list](#_Grm00117) of a [method](#_Trm00056) ([§10.6](#_Toc00441)). The formal parameter list defines the [signature](#_Trm00061) ([§3.6](#_Toc00081)) of an [instance](#_Trm00172) constructor and governs the process whereby [overload resolution](#_Trm00078) ([§7.5.2](#_Toc00227)) selects a particular [instance](#_Trm00172) constructor in an invocation.

Each of the [types](#_Trm00011) referenced in the [formal\_parameter\_list](#_Grm00117) of an [instance](#_Trm00172) constructor must be at least as [accessible](#_Trm00138) as the constructor itself ([§3.5.4](#_Toc00080)).

The optional [constructor\_initializer](#_Grm00123) specifies another [instance](#_Trm00172) constructor to invoke before executing the [statements](#_Trm00037) given in the [constructor\_body](#_Grm00123) of this [instance](#_Trm00172) constructor. This is described further in [§10.11.1](#_Toc00475).

When a constructor declaration includes an extern modifier, the constructor is said to be an ***external constructor***. Because an [external constructor](#_Trm00371) declaration provides no actual implementation, its [constructor\_body](#_Grm00123) consists of a semicolon. For all other constructors, the [constructor\_body](#_Grm00123) consists of a [block](#_Grm00071) which specifies the [statements](#_Trm00037) to initialize a new [instance](#_Trm00172) of the class. This corresponds exactly to the [block](#_Grm00071) of an [instance](#_Trm00172) [method](#_Trm00056) with a void [return type](#_Trm00060) ([§10.6.10](#_Toc00455)).

Instance constructors are not [inherited](#_Trm00136). Thus, a class has no [instance](#_Trm00172) constructors other than those actually declared in the class. If a class contains no [instance](#_Trm00172) constructor declarations, a default [instance](#_Trm00172) constructor is automatically provided ([§10.11.4](#_Toc00478)).

Instance constructors are invoked by [object\_creation\_expression](#_Grm00044)s ([§7.6.10.1](#_Toc00271)) and through [constructor\_initializer](#_Grm00123)s.

### Constructor initializers

All [instance](#_Trm00172) constructors (except those for class object) [implicit](#_Trm00197)ly include an invocation of another [instance](#_Trm00172) constructor immediately before the [constructor\_body](#_Grm00123). The constructor to [implicit](#_Trm00197)ly invoke is determined by the [constructor\_initializer](#_Grm00123):

* An [instance](#_Trm00172) constructor initializer of the form base(argument\_list) or base() causes an [instance](#_Trm00172) constructor from the direct base class to be invoked. That constructor is selected using [argument\_list](#_Grm00034) if present and the [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242). The set of candidate [instance](#_Trm00172) constructors consists of all [accessible](#_Trm00138) [instance](#_Trm00172) constructors contained in the direct base class, or the [default constructor](#_Trm00163) ([§10.11.4](#_Toc00478)), if no [instance](#_Trm00172) constructors are declared in the direct base class. If this set is empty, or if a single best [instance](#_Trm00172) constructor cannot be identified, a compile-time error occurs.
* An [instance](#_Trm00172) constructor initializer of the form this(argument-list) or this() causes an [instance](#_Trm00172) constructor from the class itself to be invoked. The constructor is selected using [argument\_list](#_Grm00034) if present and the [overload resolution](#_Trm00078) rules of [§7.5.3](#_Toc00242). The set of candidate [instance](#_Trm00172) constructors consists of all [accessible](#_Trm00138) [instance](#_Trm00172) constructors declared in the class itself. If this set is empty, or if a single best [instance](#_Trm00172) constructor cannot be identified, a compile-time error occurs. If an [instance](#_Trm00172) constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

If an [instance](#_Trm00172) constructor has no constructor initializer, a constructor initializer of the form base() is [implicit](#_Trm00197)ly provided. Thus, an [instance](#_Trm00172) constructor declaration of the form

C(...) {...}

is exactly equivalent to

C(...): base() {...}

The [scope](#_Trm00148) of the [parameters](#_Trm00059) given by the [formal\_parameter\_list](#_Grm00117) of an [instance](#_Trm00172) constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the [parameters](#_Trm00059) 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](#_Trm00172) constructor initializer cannot access the [instance](#_Trm00172) 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](#_Trm00172) member through a [simple\_name](#_Grm00036).

### Instance variable initializers

When an [instance](#_Trm00172) constructor has no constructor initializer, or it has a constructor initializer of the form base(...), that constructor [implicit](#_Trm00197)ly performs the initializations specified by the [variable\_initializer](#_Grm00115)s of the [instance](#_Trm00172) [field](#_Trm00323)s declared in its class. This corresponds to a [sequence](#_Trm00259) of assignments that are executed immediately upon entry to the constructor and before the [implicit](#_Trm00197) 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](#_Trm00037), and these assignment [statements](#_Trm00037) are executed before the invocation of the base class [instance](#_Trm00172) constructor. This ordering ensures that all [instance](#_Trm00172) [field](#_Trm00323)s are initialized by their variable initializers before any [statements](#_Trm00037) that have access to that [instance](#_Trm00172) 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](#_Trm00172) of B, the following output is produced:

x = 1, y = 0

The [value](#_Trm00209) of x is 1 because the variable initializer is executed before the base class [instance](#_Trm00172) constructor is invoked. However, the [value](#_Trm00209) of y is 0 (the [default value](#_Trm00164) 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](#_Trm00172) variable initializers and constructor initializers as [statements](#_Trm00037) that are automatically inserted before the [constructor\_body](#_Grm00123). 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](#_Trm00172) constructor declarations, a default [instance](#_Trm00172) constructor is automatically provided. That [default constructor](#_Trm00163) simply invokes the parameterless constructor of the direct base class. If the class is [abstract](#_Trm00076) then the [declared accessibility](#_Trm00140) for the [default constructor](#_Trm00163) is protected. Otherwise, the [declared accessibility](#_Trm00140) for the [default constructor](#_Trm00163) is public. Thus, the [default constructor](#_Trm00163) is always of the form

protected C(): base() {}

or

public C(): base() {}

where C is the name of the class. If [overload resolution](#_Trm00078) 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](#_Trm00163) is provided because the class contains no [instance](#_Trm00172) 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](#_Trm00172) constructors, it is not possible for classes outside the [program](#_Trm00109) text of T to derive from T or to directly create [instance](#_Trm00172)s of T. Thus, if a class contains only static [members](#_Trm00012) and isn't intended to be [instantiated](#_Trm00256), adding an empty private [instance](#_Trm00172) constructor will pr[event](#_Trm00088) 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 [method](#_Trm00056)s and [constant](#_Trm00322)s, but is not intended to be [instantiated](#_Trm00256). Therefore it declares a single empty private [instance](#_Trm00172) constructor. At least one [instance](#_Trm00172) constructor must be declared to suppress the automatic generation of a [default constructor](#_Trm00163).

### Optional [instance](#_Trm00172) constructor [parameters](#_Trm00059)

The this(...) form of constructor initializer is commonly used in conjunction with [overloading](#_Trm00077) to implement optional [instance](#_Trm00172) constructor [parameters](#_Trm00059). 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](#_Trm00172) constructors merely provide the [default value](#_Trm00164)s for the missing [arguments](#_Trm00062). Both use a this(...) constructor initializer to invoke the third [instance](#_Trm00172) constructor, which actually does the work of initializing the new [instance](#_Trm00172). The effect is that of optional constructor [parameters](#_Trm00059):

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](#_Grm00124)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](#_Grm00124) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)) and an extern modifier ([§10.6.7](#_Toc00452)).

The [identifier](#_Grm00007) of a [static\_constructor\_declaration](#_Grm00124) must name the class in which the [static constructor](#_Trm00081) is declared. If any other name is specified, a compile-time error occurs.

When a [static constructor](#_Trm00081) declaration includes an extern modifier, the [static constructor](#_Trm00081) is said to be an ***external static constructor***. Because an [external static constructor](#_Trm00373) declaration provides no actual implementation, its [static\_constructor\_body](#_Grm00124) consists of a semicolon. For all other [static constructor](#_Trm00081) declarations, the [static\_constructor\_body](#_Grm00124) consists of a [block](#_Grm00071) which specifies the [statements](#_Trm00037) to execute in order to initialize the class. This corresponds exactly to the [method\_body](#_Grm00116) of a static [method](#_Trm00056) with a void [return type](#_Trm00060) ([§10.6.10](#_Toc00455)).

Static constructors are not [inherited](#_Trm00136), and cannot be called directly.

The [static constructor](#_Trm00081) for a closed class type executes at most once in a given [application](#_Trm00124) domain. The execution of a [static constructor](#_Trm00081) is triggered by the first of the following [event](#_Trm00088)s to occur within an [application](#_Trm00124) domain:

* An [instance](#_Trm00172) of the class type is created.
* Any of the static [members](#_Trm00012) of the class type are referenced.

If a class contains the Main [method](#_Trm00056) ([§3.1](#_Toc00065)) in which execution begins, the [static constructor](#_Trm00081) for that class executes before the Main [method](#_Trm00056) is called.

To initialize a new closed class type, first a new set of static [field](#_Trm00323)s ([§10.5.1](#_Toc00432)) for that particular closed type is created. Each of the static [field](#_Trm00323)s is initialized to its [default value](#_Trm00164) ([§5.2](#_Toc00131)). Next, the static [field](#_Trm00323) initializers ([§10.5.5.1](#_Toc00439)) are executed for those static [field](#_Trm00323)s. Finally, the [static constructor](#_Trm00081) 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](#_Trm00081) is triggered by the call to A.F, and the execution of B's [static constructor](#_Trm00081) is triggered by the call to B.F.

It is possible to construct circular dependencies that allow static [field](#_Trm00323)s with variable initializers to be observed in their [default value](#_Trm00164) 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](#_Trm00056), the system first runs the initializer for B.Y, prior to class B's [static constructor](#_Trm00081). Y's initializer causes A's [static constructor](#_Trm00081) to be run because the [value](#_Trm00209) of A.X is referenced. The [static constructor](#_Trm00081) of A in turn proceeds to compute the [value](#_Trm00209) of X, and in doing so fetches the [default value](#_Trm00164) of Y, which is zero. A.X is thus initialized to 1. The process of running A's static [field](#_Trm00323) initializers and [static constructor](#_Trm00081) then completes, returning to the calculation of the initial [value](#_Trm00209) of Y, the result of which becomes 2.

Because the [static constructor](#_Trm00081) is executed exactly once for each closed constructed class type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints ([§10.1.5](#_Toc00399)). For example, the following type uses a [static constructor](#_Trm00081) 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](#_Trm00172) of a class. A [destructor](#_Trm00091) is declared using a [destructor\_declaration](#_Grm00125):

destructor\_declaration:  
 | attributes? 'extern'? '~' identifier '(' ')' destructor\_body  
 | destructor\_declaration\_unsafe  
 ;  
  
destructor\_body:  
 | block  
 | ';'  
 ;

A [destructor\_declaration](#_Grm00125) may include a set of [attributes](#_Grm00147) ([§17](#_Toc00567)).

The [identifier](#_Grm00007) of a [destructor\_declaration](#_Grm00125) must name the class in which the [destructor](#_Trm00091) is declared. If any other name is specified, a compile-time error occurs.

When a [destructor](#_Trm00091) declaration includes an extern modifier, the [destructor](#_Trm00091) is said to be an ***external destructor***. Because an external [destructor](#_Trm00091) declaration provides no actual implementation, its [destructor\_body](#_Grm00125) consists of a semicolon. For all other [destructor](#_Trm00091)s, the [destructor\_body](#_Grm00125) consists of a [block](#_Grm00071) which specifies the [statements](#_Trm00037) to execute in order to destruct an [instance](#_Trm00172) of the class. A [destructor\_body](#_Grm00125) corresponds exactly to the [method\_body](#_Grm00116) of an [instance](#_Trm00172) [method](#_Trm00056) with a void [return type](#_Trm00060) ([§10.6.10](#_Toc00455)).

Destructors are not [inherited](#_Trm00136). Thus, a class has no [destructor](#_Trm00091)s other than the one which may be declared in that class.

Since a [destructor](#_Trm00091) is required to have no [parameters](#_Trm00059), it cannot be [overloaded](#_Trm00036), so a class can have, at most, one [destructor](#_Trm00091).

Destructors are invoked automatically, and cannot be invoked [explicit](#_Trm00198)ly. An [instance](#_Trm00172) becomes [eligible](#_Trm00242) for destruction when it is no longer possible for any code to use that [instance](#_Trm00172). Execution of the [destructor](#_Trm00091) for the [instance](#_Trm00172) may occur at any time after the [instance](#_Trm00172) becomes [eligible](#_Trm00242) for destruction. When an [instance](#_Trm00172) is destructed, the [destructor](#_Trm00091)s in that [instance](#_Trm00172)'s [inheritance](#_Trm00047) chain are called, in order, from most derived to least derived. A [destructor](#_Trm00091) may be executed on any thread. For further discussion of the rules that govern when and how a [destructor](#_Trm00091) is executed, see [§3.9](#_Toc00088).

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 [destructor](#_Trm00091)s in an [inheritance](#_Trm00047) chain are called in order, from most derived to least derived.

Destructors are implemented by [overriding](#_Trm00336) the virtual [method](#_Trm00056) Finalize on System.Object. C# [program](#_Trm00109)s are not permitted to override this [method](#_Trm00056) or call it (or overrides of it) directly. For [instance](#_Trm00172), the [program](#_Trm00109)

class A  
{  
 override protected void Finalize() {} // error  
  
 public void F() {  
 this.Finalize(); // error  
 }  
}

contains two errors.

The compiler behaves as if this [method](#_Trm00056), and overrides of it, do not exist at all. Thus, this [program](#_Trm00109):

class A  
{  
 void Finalize() {} // permitted  
}

is valid, and the [method](#_Trm00056) shown [hide](#_Trm00132)s System.Object's Finalize [method](#_Trm00056).

For a discussion of the behavior when an exception is thrown from a [destructor](#_Trm00091), see [§16.3](#_Toc00565).

## Iterators

A function member ([§7.5](#_Toc00223)) implemented using an iterator [block](#_Trm00038) ([§8.2](#_Toc00350)) is called an ***iterator***.

An [iterator](#_Trm00376) [block](#_Trm00038) may be used as the body of a function member as long as the [return type](#_Trm00060) of the corresponding function member is one of the enumerator [interface](#_Trm00102)s ([§10.14.1](#_Toc00484)) or one of the enumerable [interface](#_Trm00102)s ([§10.14.2](#_Toc00485)). It can occur as a [method\_body](#_Grm00116), [operator\_body](#_Grm00122) or [accessor\_body](#_Grm00119), whereas [event](#_Trm00088)s, [instance](#_Trm00172) constructors, [static constructor](#_Trm00081)s and [destructor](#_Trm00091)s cannot be implemented as [iterator](#_Trm00376)s.

When a function member is implemented using an [iterator](#_Trm00376) [block](#_Trm00038), it is a compile-time error for the formal parameter list of the function member to specify any ref or out [parameters](#_Trm00059).

### Enumerator [interface](#_Trm00102)s

The ***enumerator interfaces*** are the non-generic [interface](#_Trm00102) System.Collections.IEnumerator and all instantiations of the generic [interface](#_Trm00102) System.Collections.Generic.IEnumerator<T>. For the sake of brevity, in this chapter these [interface](#_Trm00102)s are referenced as IEnumerator and IEnumerator<T>, respectively.

### Enumerable [interface](#_Trm00102)s

The ***enumerable interfaces*** are the non-generic [interface](#_Trm00102) System.Collections.IEnumerable and all instantiations of the generic [interface](#_Trm00102) System.Collections.Generic.IEnumerable<T>. For the sake of brevity, in this chapter these [interface](#_Trm00102)s are referenced as IEnumerable and IEnumerable<T>, respectively.

### Yield type

An [iterator](#_Trm00376) produces a [sequence](#_Trm00259) of [value](#_Trm00209)s, all of the same type. This type is called the ***yield type*** of the [iterator](#_Trm00376).

* The [yield type](#_Trm00379) of an [iterator](#_Trm00376) that returns IEnumerator or IEnumerable is object.
* The [yield type](#_Trm00379) of an [iterator](#_Trm00376) that returns IEnumerator<T> or IEnumerable<T> is T.

### Enumerator [object](#_Trm00173)s

When a function member returning an enumerator [interface](#_Trm00102) type is implemented using an [iterator](#_Trm00376) [block](#_Trm00038), invoking the function member does not immediately execute the code in the [iterator](#_Trm00376) [block](#_Trm00038). Instead, an ***enumerator object*** is created and returned. This [object](#_Trm00173) encapsulates the code specified in the [iterator](#_Trm00376) [block](#_Trm00038), and execution of the code in the [iterator](#_Trm00376) [block](#_Trm00038) occurs when the [enumerator object](#_Trm00380)'s MoveNext [method](#_Trm00056) is invoked. An [enumerator object](#_Trm00380) has the following characteristics:

* It implements IEnumerator and IEnumerator<T>, where T is the [yield type](#_Trm00379) of the [iterator](#_Trm00376).
* It implements System.IDisposable.
* It is initialized with a copy of the argument [value](#_Trm00209)s (if any) and [instance](#_Trm00172) [value](#_Trm00209) 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](#_Trm00380) is typically an [instance](#_Trm00172) of a compiler-generated enumerator class that encapsulates the code in the [iterator](#_Trm00376) [block](#_Trm00038) and implements the [enumerator interfaces](#_Trm00377), but other [method](#_Trm00056)s of implementation are possible. If an enumerator class is generated by the compiler, that class will be [nested](#_Trm00143), directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use ([§2.4.2](#_Toc00044)).

An [enumerator object](#_Trm00380) may implement more [interface](#_Trm00102)s than those specified above.

The following sections describe the exact behavior of the MoveNext, Current, and Dispose [members](#_Trm00012) of the IEnumerable and IEnumerable<T> [interface](#_Trm00102) implementations provided by an [enumerator object](#_Trm00380).

Note that [enumerator object](#_Trm00380)s do not support the IEnumerator.Reset [method](#_Trm00056). Invoking this [method](#_Trm00056) causes a System.NotSupportedException to be thrown.

#### The MoveNext [method](#_Trm00056)

The MoveNext [method](#_Trm00056) of an [enumerator object](#_Trm00380) encapsulates the code of an [iterator](#_Trm00376) [block](#_Trm00038). Invoking the MoveNext [method](#_Trm00056) executes code in the [iterator](#_Trm00376) [block](#_Trm00038) and sets the Current [property](#_Trm00348) of the [enumerator object](#_Trm00380) as appropriate. The precise action performed by MoveNext [depends on](#_Trm00306) the state of the [enumerator object](#_Trm00380) when MoveNext is invoked:

* If the state of the [enumerator object](#_Trm00380) is ***before***, invoking MoveNext:
  + Changes the state to ***running***.
  + Initializes the [parameters](#_Trm00059) (including this) of the [iterator](#_Trm00376) [block](#_Trm00038) to the argument [value](#_Trm00209)s and [instance](#_Trm00172) [value](#_Trm00209) saved when the [enumerator object](#_Trm00380) was initialized.
  + Executes the [iterator](#_Trm00376) [block](#_Trm00038) from the beginning until execution is interrupted (as described below).
* If the state of the [enumerator object](#_Trm00380) is ***running***, the result of invoking MoveNext is unspecified.
* If the state of the [enumerator object](#_Trm00380) is ***suspended***, invoking MoveNext:
  + Changes the state to ***running***.
  + Restores the [value](#_Trm00209)s of all [local variable](#_Trm00193)s and [parameters](#_Trm00059) (including this) to the [value](#_Trm00209)s saved when execution of the [iterator](#_Trm00376) [block](#_Trm00038) was last [suspended](#_Trm00383). Note that the contents of any [object](#_Trm00173)s referenced by these [variables](#_Trm00031) may have changed since the previous call to MoveNext.
  + Resumes execution of the [iterator](#_Trm00376) [block](#_Trm00038) 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](#_Trm00380) is ***after***, invoking MoveNext returns false.

When MoveNext executes the [iterator](#_Trm00376) [block](#_Trm00038), execution can be interrupted in four ways: By a yield return statement, by a yield break statement, by encountering the end of the [iterator](#_Trm00376) [block](#_Trm00038), and by an exception being thrown and propagated out of the [iterator](#_Trm00376) [block](#_Trm00038).

* When a yield return statement is encountered ([§8.14](#_Toc00376)):
  + The expression given in the statement is evaluated, [implicit](#_Trm00197)ly converted to the [yield type](#_Trm00379), and assigned to the Current [property](#_Trm00348) of the [enumerator object](#_Trm00380).
  + Execution of the [iterator](#_Trm00376) body is [suspended](#_Trm00383). The [value](#_Trm00209)s of all [local variable](#_Trm00193)s and [parameters](#_Trm00059) (including this) are saved, as is the location of this yield return statement. If the yield return statement is within one or more try [block](#_Trm00038)s, the associated finally [block](#_Trm00038)s are not executed at this time.
  + The state of the [enumerator object](#_Trm00380) is changed to ***suspended***.
  + The MoveNext [method](#_Trm00056) returns true to its caller, indicating that the iteration successfully advanced to the next [value](#_Trm00209).
* When a yield break statement is encountered ([§8.14](#_Toc00376)):
  + If the yield break statement is within one or more try [block](#_Trm00038)s, the associated finally [block](#_Trm00038)s are executed.
  + The state of the [enumerator object](#_Trm00380) is changed to ***after***.
  + The MoveNext [method](#_Trm00056) returns false to its caller, indicating that the iteration is complete.
* When the end of the [iterator](#_Trm00376) body is encountered:
  + The state of the [enumerator object](#_Trm00380) is changed to ***after***.
  + The MoveNext [method](#_Trm00056) returns false to its caller, indicating that the iteration is complete.
* When an exception is thrown and propagated out of the [iterator](#_Trm00376) [block](#_Trm00038):
  + Appropriate finally [block](#_Trm00038)s in the [iterator](#_Trm00376) body will have been executed by the [exception propagation](#_Trm00292).
  + The state of the [enumerator object](#_Trm00380) is changed to ***after***.
  + The [exception propagation](#_Trm00292) continues to the caller of the MoveNext [method](#_Trm00056).

#### The Current [property](#_Trm00348)

An [enumerator object](#_Trm00380)'s Current [property](#_Trm00348) is affected by yield return [statements](#_Trm00037) in the [iterator](#_Trm00376) [block](#_Trm00038).

When an [enumerator object](#_Trm00380) is in the ***suspended*** state, the [value](#_Trm00209) of Current is the [value](#_Trm00209) set by the previous call to MoveNext. When an [enumerator object](#_Trm00380) is in the ***before***, ***running***, or ***after*** states, the result of accessing Current is unspecified.

For an [iterator](#_Trm00376) with a [yield type](#_Trm00379) other than object, the result of accessing Current through the [enumerator object](#_Trm00380)'s IEnumerable implementation corresponds to accessing Current through the [enumerator object](#_Trm00380)'s IEnumerator<T> implementation and casting the result to object.

#### The Dispose [method](#_Trm00056)

The Dispose [method](#_Trm00056) is used to clean up the iteration by bringing the [enumerator object](#_Trm00380) to the ***after*** state.

* If the state of the [enumerator object](#_Trm00380) is ***before***, invoking Dispose changes the state to ***after***.
* If the state of the [enumerator object](#_Trm00380) is ***running***, the result of invoking Dispose is unspecified.
* If the state of the [enumerator object](#_Trm00380) is ***suspended***, invoking Dispose:
  + Changes the state to ***running***.
  + Executes any finally [block](#_Trm00038)s 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](#_Trm00376) body, the state of the [enumerator object](#_Trm00380) is set to ***after*** and the exception is propagated to the caller of the Dispose [method](#_Trm00056).
  + Changes the state to ***after***.
* If the state of the [enumerator object](#_Trm00380) is ***after***, invoking Dispose has no affect.

### Enumerable [object](#_Trm00173)s

When a function member returning an enumerable [interface](#_Trm00102) type is implemented using an [iterator](#_Trm00376) [block](#_Trm00038), invoking the function member does not immediately execute the code in the [iterator](#_Trm00376) [block](#_Trm00038). Instead, an ***enumerable object*** is created and returned. The [enumerable object](#_Trm00409)'s GetEnumerator [method](#_Trm00056) returns an [enumerator object](#_Trm00380) that encapsulates the code specified in the [iterator](#_Trm00376) [block](#_Trm00038), and execution of the code in the [iterator](#_Trm00376) [block](#_Trm00038) occurs when the [enumerator object](#_Trm00380)'s MoveNext [method](#_Trm00056) is invoked. An [enumerable object](#_Trm00409) has the following characteristics:

* It implements IEnumerable and IEnumerable<T>, where T is the [yield type](#_Trm00379) of the [iterator](#_Trm00376).
* It is initialized with a copy of the argument [value](#_Trm00209)s (if any) and [instance](#_Trm00172) [value](#_Trm00209) passed to the function member.

An [enumerable object](#_Trm00409) is typically an [instance](#_Trm00172) of a compiler-generated enumerable class that encapsulates the code in the [iterator](#_Trm00376) [block](#_Trm00038) and implements the [enumerable interfaces](#_Trm00378), but other [method](#_Trm00056)s of implementation are possible. If an enumerable class is generated by the compiler, that class will be [nested](#_Trm00143), directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use ([§2.4.2](#_Toc00044)).

An [enumerable object](#_Trm00409) may implement more [interface](#_Trm00102)s than those specified above. In particular, an [enumerable object](#_Trm00409) 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](#_Trm00409)'s GetEnumerator [method](#_Trm00056) is invoked, the [enumerable object](#_Trm00409) itself is returned. Subsequent invocations of the [enumerable object](#_Trm00409)'s GetEnumerator, if any, return a copy of the [enumerable object](#_Trm00409). Thus, each returned enumerator has its own state and changes in one enumerator will not affect another.

#### The GetEnumerator [method](#_Trm00056)

An [enumerable object](#_Trm00409) provides an implementation of the GetEnumerator [method](#_Trm00056)s of the IEnumerable and IEnumerable<T> [interface](#_Trm00102)s. The two GetEnumerator [method](#_Trm00056)s share a common implementation that acquires and returns an available [enumerator object](#_Trm00380). The [enumerator object](#_Trm00380) is initialized with the argument [value](#_Trm00209)s and [instance](#_Trm00172) [value](#_Trm00209) saved when the [enumerable object](#_Trm00409) was initialized, but otherwise the [enumerator object](#_Trm00380) functions as described in [§10.14.4](#_Toc00487).

### Implementation example

This section describes a possible implementation of [iterator](#_Trm00376)s in terms of standard C# con[structs](#_Trm00092). 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](#_Trm00056) using an [iterator](#_Trm00376). The [iterator](#_Trm00376) enumerates the [elements](#_Trm00094) 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](#_Trm00056) can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the [iterator](#_Trm00376) [block](#_Trm00038), 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](#_Trm00376) [block](#_Trm00038) is turned into a state machine and placed in the MoveNext [method](#_Trm00056) of the enumerator class. Furthermore, the [local variable](#_Trm00193) i is turned into a [field](#_Trm00323) in the [enumerator object](#_Trm00380) 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](#_Trm00056) in the example returns an [enumerable object](#_Trm00409) and is implemented using an [iterator](#_Trm00376).

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](#_Trm00056) can be translated into an instantiation of a compiler-generated enumerable class that encapsulates the code in the [iterator](#_Trm00376) [block](#_Trm00038), 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](#_Trm00378) and the [enumerator interfaces](#_Trm00377), enabling it to serve as both an enumerable and an enumerator. The first time the GetEnumerator [method](#_Trm00056) is invoked, the [enumerable object](#_Trm00409) itself is returned. Subsequent invocations of the [enumerable object](#_Trm00409)'s GetEnumerator, if any, return a copy of the [enumerable object](#_Trm00409). Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The Interlocked.CompareExchange [method](#_Trm00056) is used to ensure thread-safe operation.

The from and to [parameters](#_Trm00059) are turned into [field](#_Trm00323)s in the enumerable class. Because from is modified in the [iterator](#_Trm00376) [block](#_Trm00038), an additional \_\_from [field](#_Trm00323) is introduced to hold the initial [value](#_Trm00209) given to from in each enumerator.

The MoveNext [method](#_Trm00056) throws an InvalidOperationException if it is called when \_\_state is 0. This protects against use of the [enumerable object](#_Trm00409) as an [enumerator object](#_Trm00380) without first calling GetEnumerator.

The following example shows a simple tree class. The Tree<T> class implements its GetEnumerator [method](#_Trm00056) using an [iterator](#_Trm00376). The [iterator](#_Trm00376) enumerates the [elements](#_Trm00094) 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](#_Trm00056) can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the [iterator](#_Trm00376) [block](#_Trm00038), as shown in the following.

class Tree<T>: IEnumerable<T>  
{  
 ...  
  
 public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }  
  
 class \_\_Enumerator1 : IEnumerator<T>, IEnumerator  
 {  
 Node<T> \_\_this;  
 IEnumerator<T> \_\_left, \_\_right;  
 int \_\_state;  
 T \_\_current;  
  
 public \_\_Enumerator1(Node<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }  
  
 public T Current {  
 get { return \_\_current; }  
 }  
  
 object IEnumerator.Current {  
 get { return \_\_current; }  
 }  
  
 public bool MoveNext() {  
 try {  
 switch (\_\_state) {  
  
 case 0:  
 \_\_state = -1;  
 if (\_\_this.left == null) goto \_\_yield\_value;  
 \_\_left = \_\_this.left.GetEnumerator();  
 goto case 1;  
  
 case 1:  
 \_\_state = -2;  
 if (!\_\_left.MoveNext()) goto \_\_left\_dispose;  
 \_\_current = \_\_left.Current;  
 \_\_state = 1;  
 return true;  
  
 \_\_left\_dispose:  
 \_\_state = -1;  
 \_\_left.Dispose();  
  
 \_\_yield\_value:  
 \_\_current = \_\_this.value;  
 \_\_state = 2;  
 return true;  
  
 case 2:  
 \_\_state = -1;  
 if (\_\_this.right == null) goto \_\_end;  
 \_\_right = \_\_this.right.GetEnumerator();  
 goto case 3;  
  
 case 3:  
 \_\_state = -3;  
 if (!\_\_right.MoveNext()) goto \_\_right\_dispose;  
 \_\_current = \_\_right.Current;  
 \_\_state = 3;  
 return true;  
  
 \_\_right\_dispose:  
 \_\_state = -1;  
 \_\_right.Dispose();  
  
 \_\_end:  
 \_\_state = 4;  
 break;  
  
 }  
 }  
 finally {  
 if (\_\_state < 0) Dispose();  
 }  
 return false;  
 }  
  
 public void Dispose() {  
 try {  
 switch (\_\_state) {  
  
 case 1:  
 case -2:  
 \_\_left.Dispose();  
 break;  
  
 case 3:  
 case -3:  
 \_\_right.Dispose();  
 break;  
  
 }  
 }  
 finally {  
 \_\_state = 4;  
 }  
 }  
  
 void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

The compiler generated temporaries used in the foreach [statements](#_Trm00037) are lifted into the \_\_left and \_\_right [field](#_Trm00323)s of the [enumerator object](#_Trm00380). The \_\_state [field](#_Trm00323) of the [enumerator object](#_Trm00380) is carefully updated so that the correct Dispose() [method](#_Trm00056) will be called correctly if an exception is thrown. Note that it is not possible to write the translated code with simple foreach [statements](#_Trm00037).

## Async functions

A [method](#_Trm00056) ([§10.6](#_Toc00441)) or [anonymous function](#_Trm00253) ([§7.15](#_Toc00321)) 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](#_Trm00411) function to specify any ref or out [parameters](#_Trm00059).

The [return\_type](#_Grm00116) of an [async](#_Trm00411) [method](#_Trm00056) must be either void or a ***task type***. The [task](#_Trm00248) [types](#_Trm00011) are System.Threading.Tasks.Task and [types](#_Trm00011) constructed from System.Threading.Tasks.Task<T>. For the sake of brevity, in this chapter these [types](#_Trm00011) are referenced as Task and Task<T>, respectively. An [async](#_Trm00411) [method](#_Trm00056) returning a [task](#_Trm00248) type is said to be [task](#_Trm00248)-returning.

The exact definition of the [task](#_Trm00248) [types](#_Trm00011) is implementation [defined](#_Trm00121), but from the language's point of view a [task](#_Trm00248) type is in one of the states incomplete, succeeded or faulted. A faulted [task](#_Trm00248) records a pertinent exception. A succeeded Task<T> records a result of type T. Task [types](#_Trm00011) are [awaitable](#_Trm00249), and can therefore be the [operands](#_Trm00033) of await expressions ([§7.7.7](#_Toc00288)).

An [async](#_Trm00411) function invocation has the ability to suspend evaluation by means of await expressions ([§7.7.7](#_Toc00288)) 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](#_Trm00413) is of type System.Action, and when it is invoked, evaluation of the [async](#_Trm00411) function invocation will resume from the await expression where it left off. The ***current caller*** of an [async](#_Trm00411) function invocation is the original caller if the function invocation has never been [suspended](#_Trm00383), or the most recent caller of the [resumption delegate](#_Trm00413) otherwise.

### Evaluation of a [task](#_Trm00248)-returning [async](#_Trm00411) function

Invocation of a [task](#_Trm00248)-returning [async](#_Trm00411) function causes an [instance](#_Trm00172) of the returned [task](#_Trm00248) type to be generated. This is called the ***return task*** of the [async](#_Trm00411) function. The [task](#_Trm00248) is initially in an incomplete state.

The [async](#_Trm00411) function body is then evaluated until it is either [suspended](#_Trm00383) (by reaching an await expression) or terminates, at which point control is returned to the caller, along with the [return task](#_Trm00415).

When the body of the [async](#_Trm00411) function terminates, the [return task](#_Trm00415) 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](#_Trm00209) is recorded in the [return task](#_Trm00415), which is put into a succeeded state.
* If the function body terminates as the result of an uncaught exception ([§8.9.5](#_Toc00371)) the exception is recorded in the [return task](#_Trm00415) which is put into a faulted state.

### Evaluation of a void-returning [async](#_Trm00411) function

If the [return type](#_Trm00060) of the [async](#_Trm00411) function is void, evaluation differs from the above in the following way: Because no [task](#_Trm00248) is returned, the function instead communicates completion and exceptions to the current thread's ***synchronization context***. The exact definition of [synchronization context](#_Trm00416) is implementation-dependent, but is a representation of "where" the current thread is [running](#_Trm00382). The [synchronization context](#_Trm00416) is notified when evaluation of a void-returning [async](#_Trm00411) 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](#_Trm00411) functions are [running](#_Trm00382) under it, and to decide how to propagate exceptions coming out of them.

# Structs

Structs are similar to classes in that they represent data structures that can contain data [members](#_Trm00012) and [function members](#_Trm00079). However, unlike classes, [structs](#_Trm00092) are [value](#_Trm00209) [types](#_Trm00011) 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](#_Trm00173).

Structs are particularly useful for small data structures that have [value](#_Trm00209) semantics. Complex numbers, points in a coordinate system, or key-[value](#_Trm00209) pairs in a dictionary are all good examples of [structs](#_Trm00092). Key to these data structures is that they have few data [members](#_Trm00012), that they do not require use of [inheritance](#_Trm00047) or referential identity, and that they can be conveniently implemented using [value](#_Trm00209) semantics where assignment copies the [value](#_Trm00209) instead of the reference.

As described in [§4.1.4](#_Toc00095), the [simple types](#_Trm00020) provided by C#, such as int, double, and bool, are in fact all [struct types](#_Trm00022). Just as these pre[defined](#_Trm00121) [types](#_Trm00011) are [structs](#_Trm00092), it is also possible to use [structs](#_Trm00092) and [operator](#_Trm00090) [overloading](#_Trm00077) to implement new "primitive" [types](#_Trm00011) in the C# language. Two examples of such [types](#_Trm00011) are given at the end of this chapter ([§11.4](#_Toc00515)).

## Struct declarations

A [struct\_declaration](#_Grm00126) is a [type\_declaration](#_Grm00105) ([§9.6](#_Toc00385)) 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](#_Grm00126) consists of an optional set of [attributes](#_Grm00147) ([§17](#_Toc00567)), followed by an optional set of [struct\_modifier](#_Grm00127)s ([§11.1.1](#_Toc00499)), followed by an optional partial modifier, followed by the [keyword](#_Trm00117) struct and an [identifier](#_Grm00007) that names the struct, followed by an optional [type\_parameter\_list](#_Grm00109) specification ([§10.1.3](#_Toc00395)), followed by an optional [struct\_interfaces](#_Grm00128) specification ([§11.1.2](#_Toc00500)) ), followed by an optional [type\_parameter\_constraints\_clause](#_Grm00111)s specification ([§10.1.5](#_Toc00399)), followed by a [struct\_body](#_Grm00129) ([§11.1.4](#_Toc00502)), optionally followed by a semicolon.

### Struct modifiers

A [struct\_declaration](#_Grm00126) may optionally include a [sequence](#_Trm00259) 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 ([§10.1](#_Toc00389)).

### Partial modifier

The partial modifier indicates that this [struct\_declaration](#_Grm00126) is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in [§10.2](#_Toc00401).

### Struct [interface](#_Trm00102)s

A struct declaration may include a [struct\_interfaces](#_Grm00128) specification, in which case the struct is said to directly implement the given [interface](#_Trm00102) [types](#_Trm00011).

struct\_interfaces:  
 | ':' interface\_type\_list  
 ;

Interface implementations are discussed further in [§13.4](#_Toc00543).

### Struct body

The [struct\_body](#_Grm00129) of a struct defines the [members](#_Trm00012) of the struct.

struct\_body:  
 | '{' struct\_member\_declaration\* '}'  
 ;

## Struct [members](#_Trm00012)

The [members](#_Trm00012) of a struct consist of the [members](#_Trm00012) introduced by its [struct\_member\_declaration](#_Grm00130)s and the [members](#_Trm00012) [inherited](#_Trm00136) 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 [§11.3](#_Toc00504), the descriptions of class [members](#_Trm00012) provided in [§10.3](#_Toc00410) through [§10.14](#_Toc00483) apply to struct [members](#_Trm00012) as well.

## Class and struct differences

Structs differ from classes in several important ways:

* Structs are [value](#_Trm00209) [types](#_Trm00011) ([§11.3.1](#_Toc00505)).
* All [struct types](#_Trm00022) [implicit](#_Trm00197)ly inherit from the class System.ValueType ([§11.3.2](#_Toc00506)).
* Assignment to a variable of a struct type creates a copy of the [value](#_Trm00209) being assigned ([§11.3.3](#_Toc00507)).
* The [default value](#_Trm00164) of a struct is the [value](#_Trm00209) produced by setting all [value](#_Trm00209) type [field](#_Trm00323)s to their [default value](#_Trm00164) and all reference type [field](#_Trm00323)s to null ([§11.3.4](#_Toc00508)).
* Boxing and un[boxing](#_Trm00029) operations are used to convert between a struct type and object ([§11.3.5](#_Toc00509)).
* The meaning of this is different for [structs](#_Trm00092) ([§7.6.7](#_Toc00267)).
* Instance [field](#_Trm00323) declarations for a struct are not permitted to include variable initializers ([§11.3.7](#_Toc00511)).
* A struct is not permitted to declare a parameterless [instance](#_Trm00172) constructor ([§11.3.8](#_Toc00512)).
* A struct is not permitted to declare a [destructor](#_Trm00091) ([§11.3.9](#_Toc00513)).

### Value semantics

Structs are [value](#_Trm00209) [types](#_Trm00011) ([§4.1](#_Toc00091)) and are said to have [value](#_Trm00209) semantics. [Classes](#_Trm00044), on the other hand, are [reference types](#_Trm00019) ([§4.2](#_Toc00102)) 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](#_Trm00173). When a struct B contains an [instance](#_Trm00172) [field](#_Trm00323) 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](#_Trm00172) [field](#_Trm00323) of type Y. Given this definition, the complete set of [structs](#_Trm00092) 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](#_Trm00172) [field](#_Trm00323) 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](#_Trm00011) A, B, and C depend on each other.

With classes, it is possible for two [variables](#_Trm00031) to reference the same [object](#_Trm00173), and thus possible for operations on one variable to affect the [object](#_Trm00173) referenced by the other variable. With [structs](#_Trm00092), the [variables](#_Trm00031) each have their own copy of the data (except in the case of ref and out parameter [variables](#_Trm00031)), and it is not possible for operations on one to affect the other. Furthermore, because [structs](#_Trm00092) are not [reference types](#_Trm00019), it is not possible for [value](#_Trm00209)s 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](#_Trm00209) 10. The assignment of a to b creates a copy of the [value](#_Trm00209), 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](#_Trm00173).

### Inheritance

All [struct types](#_Trm00022) [implicit](#_Trm00197)ly inherit from the class System.ValueType, which, in turn, [inherits](#_Trm00315) from class object. A struct declaration may specify a list of implemented [interface](#_Trm00102)s, but it is not possible for a struct declaration to specify a base class.

Struct [types](#_Trm00011) are never [abstract](#_Trm00076) and are always [implicit](#_Trm00197)ly sealed. The abstract and sealed modifiers are therefore not permitted in a struct declaration.

Since [inheritance](#_Trm00047) isn't supported for [structs](#_Trm00092), the [declared accessibility](#_Trm00140) of a struct member cannot be protected or protected internal.

Function [members](#_Trm00012) in a struct cannot be abstract or virtual, and the override modifier is allowed only to override [method](#_Trm00056)s [inherited](#_Trm00136) from System.ValueType.

### Assignment

Assignment to a variable of a struct type creates a copy of the [value](#_Trm00209) being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the [object](#_Trm00173) identified by the reference.

Similar to an assignment, when a struct is passed as a [value](#_Trm00209) 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](#_Trm00348) or [indexer](#_Trm00087) of a struct is the [target](#_Trm00290) of an assignment, the [instance](#_Trm00172) expression associated with the [property](#_Trm00348) or [indexer](#_Trm00087) access must be classified as a variable. If the [instance](#_Trm00172) expression is classified as a [value](#_Trm00209), a compile-time error occurs. This is described in further detail in [§7.17.1](#_Toc00342).

### Default [value](#_Trm00209)s

As described in [§5.2](#_Toc00131), several kinds of [variables](#_Trm00031) are automatically initialized to their [default value](#_Trm00164) when they are created. For [variables](#_Trm00031) of [class types](#_Trm00024) and other [reference types](#_Trm00019), this [default value](#_Trm00164) is null. However, since [structs](#_Trm00092) are [value](#_Trm00209) [types](#_Trm00011) that cannot be null, the [default value](#_Trm00164) of a struct is the [value](#_Trm00209) produced by setting all [value](#_Trm00209) type [field](#_Trm00323)s to their [default value](#_Trm00164) and all reference type [field](#_Trm00323)s to null.

Referring to the Point struct declared above, the example

Point[] a = new Point[100];

initializes each Point in the [array](#_Trm00093) to the [value](#_Trm00209) produced by setting the x and y [field](#_Trm00323)s to zero.

The [default value](#_Trm00164) of a struct corresponds to the [value](#_Trm00209) returned by the [default constructor](#_Trm00163) of the struct ([§4.1.2](#_Toc00093)). Unlike a class, a struct is not permitted to declare a parameterless [instance](#_Trm00172) constructor. Instead, every struct [implicit](#_Trm00197)ly has a parameterless [instance](#_Trm00172) constructor which always returns the [value](#_Trm00209) that results from setting all [value](#_Trm00209) type [field](#_Trm00323)s to their [default value](#_Trm00164) and all reference type [field](#_Trm00323)s 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](#_Trm00121) [instance](#_Trm00172) constructor protects against [null value](#_Trm00165)s only where it is [explicit](#_Trm00198)ly called. In cases where a KeyValuePair variable is subject to [default value](#_Trm00164) initialization, the key and value [field](#_Trm00323)s will be null, and the struct must be prepared to handle this state.

### Boxing and un[boxing](#_Trm00029)

A [value](#_Trm00209) of a class type can be converted to type object or to an [interface](#_Trm00102) type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a [value](#_Trm00209) of type object or a [value](#_Trm00209) of an [interface](#_Trm00102) type can be converted back to a class type without changing the reference (but of course a [run-time type](#_Trm00073) check is required in this case).

Since [structs](#_Trm00092) are not [reference types](#_Trm00019), these operations are implemented differently for [struct types](#_Trm00022). When a [value](#_Trm00209) of a struct type is converted to type object or to an [interface](#_Trm00102) type that is implemented by the struct, a [boxing](#_Trm00029) operation takes place. Likewise, when a [value](#_Trm00209) of type object or a [value](#_Trm00209) of an [interface](#_Trm00102) type is converted back to a struct type, an un[boxing](#_Trm00029) operation takes place. A key difference from the same operations on [class types](#_Trm00024) is that [boxing](#_Trm00029) and un[boxing](#_Trm00029) copies the struct [value](#_Trm00209) either into or out of the boxed [instance](#_Trm00172). Thus, following a [boxing](#_Trm00029) or un[boxing](#_Trm00029) operation, changes made to the unboxed struct are not reflected in the boxed struct.

When a struct type overrides a virtual [method](#_Trm00056) [inherited](#_Trm00136) from System.Object (such as Equals, GetHashCode, or ToString), invocation of the virtual [method](#_Trm00056) through an [instance](#_Trm00172) of the struct type does not cause [boxing](#_Trm00029) to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an [instance](#_Trm00172) 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](#_Trm00109) is:

1  
2  
3

Although it is bad style for ToString to have [side effect](#_Trm00155)s, the example demonstrates that no [boxing](#_Trm00029) occurred for the three invocations of x.ToString().

Similarly, [boxing](#_Trm00029) never [implicit](#_Trm00197)ly occurs when accessing a member on a constrained type parameter. For example, suppose an [interface](#_Trm00102) ICounter contains a [method](#_Trm00056) Increment which can be used to modify a [value](#_Trm00209). If ICounter is used as a constraint, the implementation of the Increment [method](#_Trm00056) 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](#_Trm00209) in the variable x. This is not equivalent to the second call to Increment, which modifies the [value](#_Trm00209) in a boxed copy of x. Thus, the output of the [program](#_Trm00109) is:

0  
1  
1

For further details on [boxing](#_Trm00029) and un[boxing](#_Trm00029), see [§4.3](#_Toc00110).

### Meaning of this

Within an [instance](#_Trm00172) constructor or [instance](#_Trm00172) function member of a class, this is classified as a [value](#_Trm00209). Thus, while this can be used to refer to the [instance](#_Trm00172) 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](#_Trm00172) constructor of a struct, this corresponds to an out parameter of the struct type, and within an [instance](#_Trm00172) function member of a struct, this corresponds to a ref parameter of the struct type. In both cases, this is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to this or by passing this as a ref or out parameter.

### Field initializers

As described in [§11.3.4](#_Toc00508), the [default value](#_Trm00164) of a struct consists of the [value](#_Trm00209) that results from setting all [value](#_Trm00209) type [field](#_Trm00323)s to their [default value](#_Trm00164) and all reference type [field](#_Trm00323)s to null. For this reason, a struct does not permit [instance](#_Trm00172) [field](#_Trm00323) declarations to include variable initializers. This restriction applies only to [instance](#_Trm00172) [field](#_Trm00323)s. Static [field](#_Trm00323)s 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](#_Trm00172) [field](#_Trm00323) declarations include variable initializers.

### Constructors

Unlike a class, a struct is not permitted to declare a parameterless [instance](#_Trm00172) constructor. Instead, every struct [implicit](#_Trm00197)ly has a parameterless [instance](#_Trm00172) constructor which always returns the [value](#_Trm00209) that results from setting all [value](#_Trm00209) type [field](#_Trm00323)s to their [default value](#_Trm00164) and all reference type [field](#_Trm00323)s to null ([§4.1.2](#_Toc00093)). A struct can declare [instance](#_Trm00172) constructors having [parameters](#_Trm00059). 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](#_Trm00037)

Point p1 = new Point();  
Point p2 = new Point(0, 0);

both create a Point with x and y initialized to zero.

A struct [instance](#_Trm00172) constructor is not permitted to include a constructor initializer of the form base(...).

If the struct [instance](#_Trm00172) 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](#_Trm00068) ([§5.3](#_Toc00132)) at every location where the constructor returns. If the struct [instance](#_Trm00172) 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](#_Trm00068) on entry to the constructor body. Consider the [instance](#_Trm00172) 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](#_Trm00172) member function (including the set [accessors](#_Trm00083) for the properties X and Y) can be called until all [field](#_Trm00323)s of the struct being constructed have been [definitely assigned](#_Trm00068). Note, however, that if Point were a class instead of a struct, the [instance](#_Trm00172) constructor implementation would be permitted.

### Destructors

A struct is not permitted to declare a [destructor](#_Trm00091).

### Static constructors

Static constructors for [structs](#_Trm00092) follow most of the same rules as for classes. The execution of a [static constructor](#_Trm00081) for a struct type is triggered by the first of the following [event](#_Trm00088)s to occur within an [application](#_Trm00124) domain:

* A static member of the struct type is referenced.
* An [explicit](#_Trm00198)ly declared constructor of the struct type is called.

The creation of [default value](#_Trm00164)s ([§11.3.4](#_Toc00508)) of [struct types](#_Trm00022) does not trigger the [static constructor](#_Trm00081). (An example of this is the initial [value](#_Trm00209) of [elements](#_Trm00094) in an [array](#_Trm00093).)

## Struct examples

The following shows two significant examples of using struct [types](#_Trm00011) to create [types](#_Trm00011) that can be used similarly to the pre[defined](#_Trm00121) [types](#_Trm00011) 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 [value](#_Trm00209)s of the int type, plus an additional state that indicates an unknown [value](#_Trm00209). 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-[value](#_Trm00209)d logical type. The possible [value](#_Trm00209)s of this type are DBBool.True, DBBool.False, and DBBool.Null, where the Null member indicates an unknown [value](#_Trm00209). Such three-[value](#_Trm00209)d logical [types](#_Trm00011) are commonly used in databases.

using System;  
  
public struct DBBool  
{  
 // The three possible DBBool values.  
  
 public static readonly DBBool Null = new DBBool(0);  
 public static readonly DBBool False = new DBBool(-1);  
 public static readonly DBBool True = new DBBool(1);  
  
 // Private field that stores -1, 0, 1 for False, Null, True.  
  
 sbyte value;  
  
 // Private instance constructor. The value parameter must be -1, 0, or 1.  
  
 DBBool(int value) {  
 this.value = (sbyte)value;  
 }  
  
 // Properties to examine the value of a DBBool. Return true if this  
 // DBBool has the given value, false otherwise.  
  
 public bool IsNull { get { return value == 0; } }  
  
 public bool IsFalse { get { return value < 0; } }  
  
 public bool IsTrue { get { return value > 0; } }  
  
 // Implicit conversion from bool to DBBool. Maps true to DBBool.True and  
 // false to DBBool.False.  
  
 public static implicit operator DBBool(bool x) {  
 return x? True: False;  
 }  
  
 // Explicit conversion from DBBool to bool. Throws an exception if the  
 // given DBBool is Null, otherwise returns true or false.  
  
 public static explicit operator bool(DBBool x) {  
 if (x.value == 0) throw new InvalidOperationException();  
 return x.value > 0;  
 }  
  
 // Equality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.  
  
 public static DBBool operator ==(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value == y.value? True: False;  
 }  
  
 // Inequality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.  
  
 public static DBBool operator !=(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value != y.value? True: False;  
 }  
  
 // Logical negation operator. Returns True if the operand is False, Null  
 // if the operand is Null, or False if the operand is True.  
  
 public static DBBool operator !(DBBool x) {  
 return new DBBool(-x.value);  
 }  
  
 // Logical AND operator. Returns False if either operand is False,  
 // otherwise Null if either operand is Null, otherwise True.  
  
 public static DBBool operator &(DBBool x, DBBool y) {  
 return new DBBool(x.value < y.value? x.value: y.value);  
 }  
  
 // Logical OR operator. Returns True if either operand is True, otherwise  
 // Null if either operand is Null, otherwise False.  
  
 public static DBBool operator |(DBBool x, DBBool y) {  
 return new DBBool(x.value > y.value? x.value: y.value);  
 }  
  
 // Definitely true operator. Returns true if the operand is True, false  
 // otherwise.  
  
 public static bool operator true(DBBool x) {  
 return x.value > 0;  
 }  
  
 // Definitely false operator. Returns true if the operand is False, false  
 // otherwise.  
  
 public static bool operator false(DBBool x) {  
 return x.value < 0;  
 }  
  
 public override bool Equals(object obj) {  
 if (!(obj is DBBool)) return false;  
 return value == ((DBBool)obj).value;  
 }  
  
 public override int GetHashCode() {  
 return value;  
 }  
  
 public override string ToString() {  
 if (value > 0) return "DBBool.True";  
 if (value < 0) return "DBBool.False";  
 return "DBBool.Null";  
 }  
}

# Arrays

An [array](#_Trm00093) is a data structure that contains a number of [variables](#_Trm00031) which are accessed through computed indices. The [variables](#_Trm00031) contained in an [array](#_Trm00093), also called the [elements](#_Trm00094) of the [array](#_Trm00093), are all of the same type, and this type is called the [element type](#_Trm00095) of the [array](#_Trm00093).

An [array](#_Trm00093) has a [rank](#_Trm00099) which determines the number of indices associated with each [array](#_Trm00093) element. The [rank](#_Trm00099) of an [array](#_Trm00093) is also referred to as the dimensions of the [array](#_Trm00093). An [array](#_Trm00093) with a [rank](#_Trm00099) of one is called a ***single-dimensional array***. An [array](#_Trm00093) with a [rank](#_Trm00099) greater than one is called a ***multi-dimensional array***. Specific sized multi-dimensional [array](#_Trm00093)s are often referred to as two-dimensional [array](#_Trm00093)s, three-dimensional [array](#_Trm00093)s, and so on.

Each dimension of an [array](#_Trm00093) has an associated [length](#_Trm00096) which is an integral number greater than or equal to zero. The dimension [length](#_Trm00096)s are not part of the type of the [array](#_Trm00093), but rather are established when an [instance](#_Trm00172) of the [array](#_Trm00093) type is created at run-time. The [length](#_Trm00096) of a dimension determines the valid range of indices for that dimension: For a dimension of [length](#_Trm00096) N, indices can range from 0 to N - 1 inclusive. The total number of [elements](#_Trm00094) in an [array](#_Trm00093) is the product of the [length](#_Trm00096)s of each dimension in the [array](#_Trm00093). If one or more of the dimensions of an [array](#_Trm00093) have a [length](#_Trm00096) of zero, the [array](#_Trm00093) is said to be empty.

The [element type](#_Trm00095) of an [array](#_Trm00093) can be any type, including an [array](#_Trm00093) type.

## Array [types](#_Trm00011)

An [array](#_Trm00093) type is written as a [non\_array\_type](#_Grm00030) followed by one or more [rank\_specifier](#_Grm00030)s:

array\_type:  
 | non\_array\_type rank\_specifier+  
 ;  
  
non\_array\_type:  
 | type  
 ;  
  
rank\_specifier:  
 | '[' dim\_separator\* ']'  
 ;  
  
dim\_separator:  
 | ','  
 ;

A [non\_array\_type](#_Grm00030) is any [type](#_Grm00028) that is not itself an [array\_type](#_Grm00030).

The [rank](#_Trm00099) of an [array](#_Trm00093) type is given by the leftmost [rank\_specifier](#_Grm00030) in the [array\_type](#_Grm00030): A [rank\_specifier](#_Grm00030) indicates that the [array](#_Trm00093) is an [array](#_Trm00093) with a [rank](#_Trm00099) of one plus the number of "," tokens in the [rank\_specifier](#_Grm00030).

The [element type](#_Trm00095) of an [array](#_Trm00093) type is the type that results from deleting the leftmost [rank\_specifier](#_Grm00030):

* An [array](#_Trm00093) type of the form T[R] is an [array](#_Trm00093) with [rank](#_Trm00099) R and a non-[array](#_Trm00093) [element type](#_Trm00095) T.
* An [array](#_Trm00093) type of the form T[R][R1]...[Rn] is an [array](#_Trm00093) with [rank](#_Trm00099) R and an [element type](#_Trm00095) T[R1]...[Rn].

In effect, the [rank\_specifier](#_Grm00030)s are read from left to right [before](#_Trm00381) the final non-[array](#_Trm00093) [element type](#_Trm00095). The type int[][,,][,] is a single-dimensional [array](#_Trm00093) of three-dimensional [array](#_Trm00093)s of two-dimensional [array](#_Trm00093)s of int.

At run-time, a [value](#_Trm00209) of an [array](#_Trm00093) type can be null or a reference to an [instance](#_Trm00172) of that [array](#_Trm00093) type.

### The System.Array type

The type System.Array is the [abstract](#_Trm00076) base type of all [array](#_Trm00093) [types](#_Trm00011). An [implicit](#_Trm00197) reference [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) exists from any [array](#_Trm00093) type to System.Array, and an [explicit](#_Trm00198) reference [conversion](#_Trm00196) ([§6.2.4](#_Toc00185)) exists from System.Array to any [array](#_Trm00093) type. Note that System.Array is not itself an [array\_type](#_Grm00030). Rather, it is a [class\_type](#_Grm00030) from which all [array\_type](#_Grm00030)s are derived.

At run-time, a [value](#_Trm00209) of type System.Array can be null or a reference to an [instance](#_Trm00172) of any [array](#_Trm00093) type.

### Arrays and the generic IList [interface](#_Trm00102)

A one-dimensional [array](#_Trm00093) T[] implements the [interface](#_Trm00102) System.Collections.Generic.IList<T> (IList<T> for short) and its base [interface](#_Trm00102)s. Accordingly, there is an [implicit](#_Trm00197) [conversion](#_Trm00196) from T[] to IList<T> and its base [interface](#_Trm00102)s. In addition, if there is an [implicit](#_Trm00197) reference [conversion](#_Trm00196) from S to T then S[] implements IList<T> and there is an [implicit](#_Trm00197) reference [conversion](#_Trm00196) from S[] to IList<T> and its base [interface](#_Trm00102)s ([§6.1.6](#_Toc00174)). If there is an [explicit](#_Trm00198) reference [conversion](#_Trm00196) from S to T then there is an [explicit](#_Trm00198) reference [conversion](#_Trm00196) from S[] to IList<T> and its base [interface](#_Trm00102)s ([§6.2.4](#_Toc00185)). 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](#_Trm00196) from object[] to IList<string> is an [explicit](#_Trm00198) [conversion](#_Trm00196), not [implicit](#_Trm00197). The cast (IList<string>)oa1 will cause an exception to be thrown at run-time since oa1 [references](#_Trm00160) an object[] and not a string[]. However the cast (IList<string>)oa2 will not cause an exception to be thrown since oa2 [references](#_Trm00160) a string[].

Whenever there is an [implicit](#_Trm00197) or [explicit](#_Trm00198) reference [conversion](#_Trm00196) from S[] to IList<T>, there is also an [explicit](#_Trm00198) reference [conversion](#_Trm00196) from IList<T> and its base [interface](#_Trm00102)s to S[] ([§6.2.4](#_Toc00185)).

When an [array](#_Trm00093) type S[] implements IList<T>, some of the [members](#_Trm00012) of the implemented [interface](#_Trm00102) may throw exceptions. The precise behavior of the implementation of the [interface](#_Trm00102) is beyond the [scope](#_Trm00148) of this specification.

## Array creation

Array [instance](#_Trm00172)s are created by [array\_creation\_expression](#_Grm00047)s ([§7.6.10.4](#_Toc00274)) or by [field](#_Trm00323) or [local variable](#_Trm00193) declarations that include an [array\_initializer](#_Grm00132) ([§12.6](#_Toc00526)).

When an [array](#_Trm00093) [instance](#_Trm00172) is created, the [rank](#_Trm00099) and [length](#_Trm00096) of each dimension are established and then remain [constant](#_Trm00322) for the entire lifetime of the [instance](#_Trm00172). In other words, it is not possible to change the [rank](#_Trm00099) of an existing [array](#_Trm00093) [instance](#_Trm00172), nor is it possible to resize its dimensions.

An [array](#_Trm00093) [instance](#_Trm00172) is always of an [array](#_Trm00093) type. The System.Array type is an [abstract](#_Trm00076) type that cannot be [instantiated](#_Trm00256).

Elements of [array](#_Trm00093)s created by [array\_creation\_expression](#_Grm00047)s are always initialized to their [default value](#_Trm00164) ([§5.2](#_Toc00131)).

## Array element access

Array [elements](#_Trm00094) are accessed using [element\_access](#_Grm00040) expressions ([§7.6.6.1](#_Toc00265)) of the form A[I1, I2, ..., In], where A is an expression of an [array](#_Trm00093) type and each Ix is an expression of type int, uint, long, ulong, or can be [implicit](#_Trm00197)ly converted to one or more of these [types](#_Trm00011). The result of an [array](#_Trm00093) element access is a variable, namely the [array](#_Trm00093) element selected by the indices.

The [elements](#_Trm00094) of an [array](#_Trm00093) can be enumerated using a foreach statement ([§8.8.4](#_Toc00365)).

## Array [members](#_Trm00012)

Every [array](#_Trm00093) type [inherits](#_Trm00315) the [members](#_Trm00012) declared by the System.Array type.

## Array covariance

For any two [reference\_type](#_Grm00030)s A and B, if an [implicit](#_Trm00197) reference [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) or [explicit](#_Trm00198) reference [conversion](#_Trm00196) ([§6.2.4](#_Toc00185)) exists from A to B, then the same reference [conversion](#_Trm00196) also exists from the [array](#_Trm00093) type A[R] to the [array](#_Trm00093) type B[R], where R is any given [rank\_specifier](#_Grm00030) (but the same for both [array](#_Trm00093) [types](#_Trm00011)). This relationship is known as ***array covariance***. Array covariance in particular means that a [value](#_Trm00209) of an [array](#_Trm00093) type A[R] may actually be a reference to an [instance](#_Trm00172) of an [array](#_Trm00093) type B[R], provided an [implicit](#_Trm00197) reference [conversion](#_Trm00196) exists from B to A.

Because of [array](#_Trm00093) covariance, assignments to [elements](#_Trm00094) of reference type [array](#_Trm00093)s include a run-time check which ensures that the [value](#_Trm00209) being assigned to the [array](#_Trm00093) element is actually of a permitted type ([§7.17.1](#_Toc00342)). 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](#_Trm00056) [implicit](#_Trm00197)ly includes a run-time check which ensures that the [object](#_Trm00173) referenced by value is either null or an [instance](#_Trm00172) that is compatible with the actual [element type](#_Trm00095) 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](#_Trm00093).

Array covariance specifically does not extend to [array](#_Trm00093)s of [value\_type](#_Grm00029)s. For example, no [conversion](#_Trm00196) exists that permits an int[] to be treated as an object[].

## Array initializers

Array initializers may be specified in [field](#_Trm00323) declarations ([§10.5](#_Toc00431)), [local variable](#_Trm00193) declarations ([§8.5.1](#_Toc00355)), and [array](#_Trm00093) creation expressions ([§7.6.10.4](#_Toc00274)):

array\_initializer:  
 | '{' variable\_initializer\_list? '}'  
 | '{' variable\_initializer\_list ',' '}'  
 ;  
  
variable\_initializer\_list:  
 | variable\_initializer ( ',' variable\_initializer )\*  
 ;  
  
variable\_initializer:  
 | expression  
 | array\_initializer  
 ;

An [array](#_Trm00093) initializer consists of a [sequence](#_Trm00259) 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](#_Trm00093), a [nested](#_Trm00143) [array](#_Trm00093) initializer.

The context in which an [array](#_Trm00093) initializer is used determines the type of the [array](#_Trm00093) being initialized. In an [array](#_Trm00093) creation expression, the [array](#_Trm00093) type immediately precedes the initializer, or is inferred from the expressions in the [array](#_Trm00093) initializer. In a [field](#_Trm00323) or variable declaration, the [array](#_Trm00093) type is the type of the [field](#_Trm00323) or variable being declared. When an [array](#_Trm00093) initializer is used in a [field](#_Trm00323) or variable declaration, such as:

int[] a = {0, 2, 4, 6, 8};

it is simply shorthand for an equivalent [array](#_Trm00093) creation expression:

int[] a = new int[] {0, 2, 4, 6, 8};

For a single-dimensional [array](#_Trm00093), the [array](#_Trm00093) initializer must consist of a [sequence](#_Trm00259) of expressions that are assignment compatible with the [element type](#_Trm00095) of the [array](#_Trm00093). The expressions initialize [array](#_Trm00093) [elements](#_Trm00094) in increasing order, starting with the element at index zero. The number of expressions in the [array](#_Trm00093) initializer determines the [length](#_Trm00096) of the [array](#_Trm00093) [instance](#_Trm00172) being created. For example, the [array](#_Trm00093) initializer above creates an int[] [instance](#_Trm00172) of [length](#_Trm00096) 5 and then initializes the [instance](#_Trm00172) with the following [value](#_Trm00209)s:

a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;

For a multi-dimensional [array](#_Trm00093), the [array](#_Trm00093) initializer must have as many levels of nesting as there are dimensions in the [array](#_Trm00093). The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The [length](#_Trm00096) of each dimension of the [array](#_Trm00093) is determined by the number of [elements](#_Trm00094) at the corresponding nesting level in the [array](#_Trm00093) initializer. For each [nested](#_Trm00143) [array](#_Trm00093) initializer, the number of [elements](#_Trm00094) must be the same as the other [array](#_Trm00093) initializers at the same level. The example:

int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};

creates a two-dimensional [array](#_Trm00093) with a [length](#_Trm00096) of five for the leftmost dimension and a [length](#_Trm00096) of two for the rightmost dimension:

int[,] b = new int[5, 2];

and then initializes the [array](#_Trm00093) [instance](#_Trm00172) with the following [value](#_Trm00209)s:

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](#_Trm00096) zero, the subsequent dimensions are assumed to also have [length](#_Trm00096) zero. The example:

int[,] c = {};

creates a two-dimensional [array](#_Trm00093) with a [length](#_Trm00096) of zero for both the leftmost and the rightmost dimension:

int[,] c = new int[0, 0];

When an [array](#_Trm00093) creation expression includes both [explicit](#_Trm00198) dimension [length](#_Trm00096)s and an [array](#_Trm00093) initializer, the [length](#_Trm00096)s must be [constant](#_Trm00322) expressions and the number of [elements](#_Trm00094) at each nesting level must match the corresponding dimension [length](#_Trm00096). 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](#_Trm00096) expression is not a [constant](#_Trm00322), and the initializer for z results in a compile-time error because the [length](#_Trm00096) and the number of [elements](#_Trm00094) in the initializer do not agree.

# Interfaces

An [interface](#_Trm00102) defines a contract. A class or struct that implements an [interface](#_Trm00102) must adhere to its contract. An [interface](#_Trm00102) may inherit from multiple base [interface](#_Trm00102)s, and a class or struct may implement multiple [interface](#_Trm00102)s.

Interfaces can contain [method](#_Trm00056)s, properties, [event](#_Trm00088)s, and [indexer](#_Trm00087)s. The [interface](#_Trm00102) itself does not provide implementations for the [members](#_Trm00012) that it defines. The [interface](#_Trm00102) merely specifies the [members](#_Trm00012) that must be supplied by classes or [structs](#_Trm00092) that implement the [interface](#_Trm00102).

## Interface declarations

An [interface\_declaration](#_Grm00133) is a [type\_declaration](#_Grm00105) ([§9.6](#_Toc00385)) that declares a new [interface](#_Trm00102) type.

interface\_declaration:  
 | attributes? interface\_modifier\* 'partial'? 'interface'  
 identifier variant\_type\_parameter\_list? interface\_base?  
 type\_parameter\_constraints\_clause\* interface\_body ';'?  
 ;

An [interface\_declaration](#_Grm00133) consists of an optional set of [attributes](#_Grm00147) ([§17](#_Toc00567)), followed by an optional set of [interface\_modifier](#_Grm00134)s ([§13.1.1](#_Toc00529)), followed by an optional partial modifier, followed by the [keyword](#_Trm00117) interface and an [identifier](#_Grm00007) that names the [interface](#_Trm00102), followed by an optional [variant\_type\_parameter\_list](#_Grm00135) specification ([§13.1.3](#_Toc00531)), followed by an optional [interface\_base](#_Grm00136) specification ([§13.1.4](#_Toc00534)), followed by an optional [type\_parameter\_constraints\_clause](#_Grm00111)s specification ([§10.1.5](#_Toc00399)), followed by an [interface\_body](#_Grm00137) ([§13.1.5](#_Toc00535)), optionally followed by a semicolon.

### Interface modifiers

An [interface\_declaration](#_Grm00133) may optionally include a [sequence](#_Trm00259) of [interface](#_Trm00102) 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](#_Trm00102) declaration.

The new modifier is only permitted on [interface](#_Trm00102)s [defined](#_Trm00121) within a class. It specifies that the [interface](#_Trm00102) [hide](#_Trm00132)s an [inherited](#_Trm00136) member by the same name, as described in [§10.3.4](#_Toc00414).

The public, protected, internal, and private modifiers control the accessibility of the [interface](#_Trm00102). Depending on the context in which the [interface](#_Trm00102) declaration occurs, only some of these modifiers may be permitted ([§3.5.1](#_Toc00077)).

### Partial modifier

The partial modifier indicates that this [interface\_declaration](#_Grm00133) is a partial type declaration. Multiple partial [interface](#_Trm00102) declarations with the same name within an enclosing namespace or type declaration combine to form one [interface](#_Trm00102) declaration, following the rules specified in [§10.2](#_Toc00401).

### Variant type parameter lists

Variant type parameter lists can only occur on [interface](#_Trm00102) and [delegate type](#_Trm00107)s. The difference from ordinary [type\_parameter\_list](#_Grm00109)s is the optional [variance\_annotation](#_Grm00135) 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](#_Trm00422), Y is [contravariant](#_Trm00423) and Z is [invariant](#_Trm00424).

#### Variance safety

The occurrence of variance annotations in the type parameter list of a type restricts the places where [types](#_Trm00011) can occur within the type declaration.

A type T is ***output-unsafe*** if one of the following holds:

* T is a [contravariant](#_Trm00423) type parameter
* T is an [array](#_Trm00093) type with an [output-unsafe](#_Trm00425) [element type](#_Trm00095)
* T is an [interface](#_Trm00102) or [delegate type](#_Trm00107) 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](#_Trm00422) or [invariant](#_Trm00424) and Ai is [output-unsafe](#_Trm00425).
  + Xi is [contravariant](#_Trm00423) or [invariant](#_Trm00424) and Ai is input-safe.

A type T is ***input-unsafe*** if one of the following holds:

* T is a [covariant](#_Trm00422) type parameter
* T is an [array](#_Trm00093) type with an [input-unsafe](#_Trm00426) [element type](#_Trm00095)
* T is an [interface](#_Trm00102) or [delegate type](#_Trm00107) 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](#_Trm00422) or [invariant](#_Trm00424) and Ai is [input-unsafe](#_Trm00426).
  + Xi is [contravariant](#_Trm00423) or [invariant](#_Trm00424) and Ai is [output-unsafe](#_Trm00425).

Intuitively, an [output-unsafe](#_Trm00425) type is prohibited in an output position, and an [input-unsafe](#_Trm00426) type is prohibited in an input position.

A type is ***output-safe*** if it is not [output-unsafe](#_Trm00425), and ***input-safe*** if it is not [input-unsafe](#_Trm00426).

#### Variance [conversion](#_Trm00196)

The purpose of variance annotations is to provide for more lenient (but still type safe) [conversion](#_Trm00196)s to [interface](#_Trm00102) and [delegate type](#_Trm00107)s. To this end the definitions of [implicit](#_Trm00197) ([§6.1](#_Toc00168)) and [explicit](#_Trm00198) [conversion](#_Trm00196)s ([§6.2](#_Toc00181)) make use of the notion of variance-convertibility, which is [defined](#_Trm00121) as follows:

A type T<A1,...,An> is variance-convertible to a type T<B1,...,Bn> if T is either an [interface](#_Trm00102) or a [delegate type](#_Trm00107) declared with the variant type [parameters](#_Trm00059) T<X1,...,Xn>, and for each variant type parameter Xi one of the following holds:

* Xi is [covariant](#_Trm00422) and an [implicit](#_Trm00197) reference or identity [conversion](#_Trm00196) exists from Ai to Bi
* Xi is [contravariant](#_Trm00423) and an [implicit](#_Trm00197) reference or identity [conversion](#_Trm00196) exists from Bi to Ai
* Xi is [invariant](#_Trm00424) and an identity [conversion](#_Trm00196) exists from Ai to Bi

### Base [interface](#_Trm00102)s

An [interface](#_Trm00102) can inherit from zero or more [interface](#_Trm00102) [types](#_Trm00011), which are called the ***explicit base interfaces*** of the [interface](#_Trm00102). When an [interface](#_Trm00102) has one or more [explicit](#_Trm00198) base [interface](#_Trm00102)s, then in the declaration of that [interface](#_Trm00102), the [interface](#_Trm00102) identifier is followed by a colon and a comma separated list of base [interface](#_Trm00102) [types](#_Trm00011).

interface\_base:  
 | ':' interface\_type\_list  
 ;

For a constructed [interface](#_Trm00102) type, the [explicit](#_Trm00198) base [interface](#_Trm00102)s are formed by taking the [explicit](#_Trm00198) base [interface](#_Trm00102) declarations on the generic type declaration, and substituting, for each [type\_parameter](#_Grm00032) in the base [interface](#_Trm00102) declaration, the corresponding [type\_argument](#_Grm00031) of the [constructed type](#_Trm00178).

The [explicit](#_Trm00198) base [interface](#_Trm00102)s of an [interface](#_Trm00102) must be at least as [accessible](#_Trm00138) as the [interface](#_Trm00102) itself ([§3.5.4](#_Toc00080)). For example, it is a compile-time error to specify a private or internal [interface](#_Trm00102) in the [interface\_base](#_Grm00136) of a public [interface](#_Trm00102).

It is a compile-time error for an [interface](#_Trm00102) to directly or indirectly inherit from itself.

The ***base interfaces*** of an [interface](#_Trm00102) are the [explicit](#_Trm00198) [base interfaces](#_Trm00430) and their [base interfaces](#_Trm00430). In other words, the set of [base interfaces](#_Trm00430) is the complete transitive closure of the [explicit](#_Trm00198) [base interfaces](#_Trm00430), their [explicit](#_Trm00198) [base interfaces](#_Trm00430), and so on. An [interface](#_Trm00102) [inherits](#_Trm00315) all [members](#_Trm00012) of its [base interfaces](#_Trm00430). 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](#_Trm00430) of IComboBox are IControl, ITextBox, and IListBox.

In other words, the IComboBox [interface](#_Trm00102) above [inherits](#_Trm00315) [members](#_Trm00012) SetText and SetItems as well as Paint.

Every base [interface](#_Trm00102) of an [interface](#_Trm00102) must be [output-safe](#_Trm00427) ([§13.1.3.1](#_Toc00532)). A class or struct that implements an [interface](#_Trm00102) also [implicit](#_Trm00197)ly implements all of the [interface](#_Trm00102)'s [base interfaces](#_Trm00430).

### Interface body

The [interface\_body](#_Grm00137) of an [interface](#_Trm00102) defines the [members](#_Trm00012) of the [interface](#_Trm00102).

interface\_body:  
 | '{' interface\_member\_declaration\* '}'  
 ;

## Interface [members](#_Trm00012)

The [members](#_Trm00012) of an [interface](#_Trm00102) are the [members](#_Trm00012) [inherited](#_Trm00136) from the [base interfaces](#_Trm00430) and the [members](#_Trm00012) declared by the [interface](#_Trm00102) itself.

interface\_member\_declaration:  
 | interface\_method\_declaration  
 | interface\_property\_declaration  
 | interface\_event\_declaration  
 | interface\_indexer\_declaration  
 ;

An [interface](#_Trm00102) declaration may declare zero or more [members](#_Trm00012). The [members](#_Trm00012) of an [interface](#_Trm00102) must be [method](#_Trm00056)s, properties, [event](#_Trm00088)s, or [indexer](#_Trm00087)s. An [interface](#_Trm00102) cannot contain [constant](#_Trm00322)s, [field](#_Trm00323)s, [operator](#_Trm00090)s, [instance](#_Trm00172) constructors, [destructor](#_Trm00091)s, or [types](#_Trm00011), nor can an [interface](#_Trm00102) contain static [members](#_Trm00012) of any kind.

All [interface](#_Trm00102) [members](#_Trm00012) [implicit](#_Trm00197)ly have public access. It is a compile-time error for [interface](#_Trm00102) member declarations to include any modifiers. In particular, [interface](#_Trm00102)s [members](#_Trm00012) 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](#_Trm00102) that contains one each of the possible kinds of [members](#_Trm00012): A [method](#_Trm00056), a [property](#_Trm00348), an [event](#_Trm00088), and an [indexer](#_Trm00087).

An [interface\_declaration](#_Grm00133) creates a new [declaration space](#_Trm00130) ([§3.3](#_Toc00067)), and the [interface\_member\_declaration](#_Grm00138)s immediately contained by the [interface\_declaration](#_Grm00133) introduce new [members](#_Trm00012) into this [declaration space](#_Trm00130). The following rules apply to [interface\_member\_declaration](#_Grm00138)s:

* The name of a [method](#_Trm00056) must differ from the names of all properties and [event](#_Trm00088)s declared in the same [interface](#_Trm00102). In addition, the [signature](#_Trm00061) ([§3.6](#_Toc00081)) of a [method](#_Trm00056) must differ from the [signature](#_Trm00061)s of all other [method](#_Trm00056)s declared in the same [interface](#_Trm00102), and two [method](#_Trm00056)s declared in the same [interface](#_Trm00102) may not have [signature](#_Trm00061)s that differ solely by ref and out.
* The name of a [property](#_Trm00348) or [event](#_Trm00088) must differ from the names of all other [members](#_Trm00012) declared in the same [interface](#_Trm00102).
* The [signature](#_Trm00061) of an [indexer](#_Trm00087) must differ from the [signature](#_Trm00061)s of all other [indexer](#_Trm00087)s declared in the same [interface](#_Trm00102).

The [inherited](#_Trm00136) [members](#_Trm00012) of an [interface](#_Trm00102) are specifically not part of the [declaration space](#_Trm00130) of the [interface](#_Trm00102). Thus, an [interface](#_Trm00102) is allowed to declare a member with the same name or [signature](#_Trm00061) as an [inherited](#_Trm00136) member. When this occurs, the derived [interface](#_Trm00102) member is said to [hide](#_Trm00132) the base [interface](#_Trm00102) member. Hiding an [inherited](#_Trm00136) 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](#_Trm00102) member must include a new modifier to indicate that the derived member is intended to [hide](#_Trm00132) the base member. This topic is discussed further in [§3.7.1.2](#_Toc00085).

If a new modifier is included in a declaration that doesn't [hide](#_Trm00132) an [inherited](#_Trm00136) member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

Note that the [members](#_Trm00012) in class object are not, strictly speaking, [members](#_Trm00012) of any [interface](#_Trm00102) ([§13.2](#_Toc00536)). However, the [members](#_Trm00012) in class object are available via member lookup in any [interface](#_Trm00102) type ([§7.4](#_Toc00221)).

### Interface [method](#_Trm00056)s

Interface [method](#_Trm00056)s are declared using [interface\_method\_declaration](#_Grm00139)s:

interface\_method\_declaration:  
 | attributes? 'new'? return\_type identifier type\_parameter\_list  
 '(' formal\_parameter\_list? ')' type\_parameter\_constraints\_clause\* ';'  
 ;

The [attributes](#_Grm00147), [return\_type](#_Grm00116), [identifier](#_Grm00007), and [formal\_parameter\_list](#_Grm00117) of an [interface](#_Trm00102) [method](#_Trm00056) declaration have the same meaning as those of a [method](#_Trm00056) declaration in a class ([§10.6](#_Toc00441)). An [interface](#_Trm00102) [method](#_Trm00056) declaration is not permitted to specify a [method](#_Trm00056) body, and the declaration therefore always ends with a semicolon.

Each formal parameter type of an [interface](#_Trm00102) [method](#_Trm00056) must be [input-safe](#_Trm00428) ([§13.1.3.1](#_Toc00532)), and the [return type](#_Trm00060) must be either void or [output-safe](#_Trm00427). Furthermore, each class type constraint, [interface](#_Trm00102) type constraint and type parameter constraint on any type parameter of the [method](#_Trm00056) must be [input-safe](#_Trm00428).

These rules ensure that any [covariant](#_Trm00422) or [contravariant](#_Trm00423) usage of the [interface](#_Trm00102) remains [types](#_Trm00011)afe. 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](#_Trm00428).

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\_declaration](#_Grm00140)s:

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](#_Grm00147), [type](#_Grm00028), and [identifier](#_Grm00007) of an [interface](#_Trm00102) [property](#_Trm00348) declaration have the same meaning as those of a [property](#_Trm00348) declaration in a class ([§10.7](#_Toc00457)).

The [accessors](#_Trm00083) of an [interface](#_Trm00102) [property](#_Trm00348) declaration correspond to the [accessors](#_Trm00083) of a class [property](#_Trm00348) declaration ([§10.7.2](#_Toc00459)), except that the accessor body must always be a semicolon. Thus, the [accessors](#_Trm00083) simply indicate whether the [property](#_Trm00348) is [read-write](#_Trm00354), [read-only](#_Trm00355), or [write-only](#_Trm00356).

The type of an [interface](#_Trm00102) [property](#_Trm00348) must be [output-safe](#_Trm00427) if there is a get accessor, and must be [input-safe](#_Trm00428) if there is a set accessor.

### Interface [event](#_Trm00088)s

Interface [event](#_Trm00088)s are declared using [interface\_event\_declaration](#_Grm00141)s:

interface\_event\_declaration:  
 | attributes? 'new'? 'event' type identifier ';'  
 ;

The [attributes](#_Grm00147), [type](#_Grm00028), and [identifier](#_Grm00007) of an [interface](#_Trm00102) [event](#_Trm00088) declaration have the same meaning as those of an [event](#_Trm00088) declaration in a class ([§10.8](#_Toc00463)).

The type of an [interface](#_Trm00102) [event](#_Trm00088) must be [input-safe](#_Trm00428).

### Interface [indexer](#_Trm00087)s

Interface [indexer](#_Trm00087)s are declared using [interface\_indexer\_declaration](#_Grm00142)s:

interface\_indexer\_declaration:  
 | attributes? 'new'? type 'this' '[' formal\_parameter\_list ']' '{' interface\_accessors '}'  
 ;

The [attributes](#_Grm00147), [type](#_Grm00028), and [formal\_parameter\_list](#_Grm00117) of an [interface](#_Trm00102) [indexer](#_Trm00087) declaration have the same meaning as those of an [indexer](#_Trm00087) declaration in a class ([§10.9](#_Toc00468)).

The [accessors](#_Trm00083) of an [interface](#_Trm00102) [indexer](#_Trm00087) declaration correspond to the [accessors](#_Trm00083) of a class [indexer](#_Trm00087) declaration ([§10.9](#_Toc00468)), except that the accessor body must always be a semicolon. Thus, the [accessors](#_Trm00083) simply indicate whether the [indexer](#_Trm00087) is [read-write](#_Trm00354), [read-only](#_Trm00355), or [write-only](#_Trm00356).

All the formal parameter [types](#_Trm00011) of an [interface](#_Trm00102) [indexer](#_Trm00087) must be [input-safe](#_Trm00428) . In addition, any out or ref formal parameter [types](#_Trm00011) must also be [output-safe](#_Trm00427). Note that even out [parameters](#_Trm00059) are required to be [input-safe](#_Trm00428), due to a limitiation of the underlying execution platform.

The type of an [interface](#_Trm00102) [indexer](#_Trm00087) must be [output-safe](#_Trm00427) if there is a get accessor, and must be [input-safe](#_Trm00428) if there is a set accessor.

### Interface member access

Interface [members](#_Trm00012) are accessed through member access ([§7.6.4](#_Toc00257)) and [indexer](#_Trm00087) access ([§7.6.6.2](#_Toc00266)) expressions of the form I.M and I[A], where I is an [interface](#_Trm00102) type, M is a [method](#_Trm00056), [property](#_Trm00348), or [event](#_Trm00088) of that [interface](#_Trm00102) type, and A is an [indexer](#_Trm00087) argument list.

For [interface](#_Trm00102)s that are strictly single-[inheritance](#_Trm00047) (each [interface](#_Trm00102) in the [inheritance](#_Trm00047) chain has exactly zero or one direct base [interface](#_Trm00102)), the effects of the member lookup ([§7.4](#_Toc00221)), [method](#_Trm00056) invocation ([§7.6.5.1](#_Toc00261)), and [indexer](#_Trm00087) access ([§7.6.6.2](#_Toc00266)) rules are exactly the same as for classes and [structs](#_Trm00092): More derived [members](#_Trm00012) [hide](#_Trm00132) less derived [members](#_Trm00012) with the same name or [signature](#_Trm00061). However, for multiple-[inheritance](#_Trm00047) [interface](#_Trm00102)s, ambiguities can occur when two or more unrelated [base interfaces](#_Trm00430) declare [members](#_Trm00012) with the same name or [signature](#_Trm00061). This section shows several examples of such situations. In all cases, [explicit](#_Trm00198) 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](#_Trm00037) cause compile-time errors because the member lookup ([§7.4](#_Toc00221)) of Count in IListCounter is ambiguous. As illustrated by the example, the ambiguity is resolved by casting x to the appropriate base [interface](#_Trm00102) type. Such casts have no run-time costs—they merely consist of viewing the [instance](#_Trm00172) 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](#_Trm00078) rules of [§7.5.3](#_Toc00242). Similarly the invocation n.Add(1.0) selects IDouble.Add. When [explicit](#_Trm00198) casts are inserted, there is only one candidate [method](#_Trm00056), 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](#_Trm00150) by the ILeft.F member. The invocation d.F(1) thus selects ILeft.F, even though IBase.F appears to not be [hidden](#_Trm00150) in the access path that leads through IRight.

The intuitive rule for hiding in multiple-[inheritance](#_Trm00047) [interface](#_Trm00102)s is simply this: If a member is [hidden](#_Trm00150) in any access path, it is [hidden](#_Trm00150) in all access paths. Because the access path from IDerived to ILeft to IBase [hide](#_Trm00132)s IBase.F, the member is also [hidden](#_Trm00150) in the access path from IDerived to IRight to IBase.

## Fully qualified [interface](#_Trm00102) member names

An [interface](#_Trm00102) member is sometimes referred to by its ***fully qualified name***. The [fully qualified name](#_Trm00153) of an [interface](#_Trm00102) member consists of the name of the [interface](#_Trm00102) in which the member is declared, followed by a dot, followed by the name of the member. The [fully qualified name](#_Trm00153) of a member [references](#_Trm00160) the [interface](#_Trm00102) 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](#_Trm00153) of Paint is IControl.Paint and the [fully qualified name](#_Trm00153) of SetText is ITextBox.SetText.

In the example above, it is not possible to refer to Paint as ITextBox.Paint.

When an [interface](#_Trm00102) is part of a namespace, the [fully qualified name](#_Trm00153) of an [interface](#_Trm00102) member includes the namespace name. For example

namespace System  
{  
 public interface ICloneable  
 {  
 object Clone();  
 }  
}

Here, the [fully qualified name](#_Trm00153) of the Clone [method](#_Trm00056) is System.ICloneable.Clone.

## Interface implementations

Interfaces may be implemented by classes and [structs](#_Trm00092). To indicate that a class or struct directly implements an [interface](#_Trm00102), the [interface](#_Trm00102) 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](#_Trm00102) also directly implements all of the [interface](#_Trm00102)'s [base interfaces](#_Trm00430) [implicit](#_Trm00197)ly. This is true even if the class or struct doesn't [explicit](#_Trm00198)ly list all [base interfaces](#_Trm00430) 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](#_Trm00102), all classes derived from C also implement the [interface](#_Trm00102) [implicit](#_Trm00197)ly. The [base interfaces](#_Trm00430) specified in a class declaration can be constructed [interface](#_Trm00102) [types](#_Trm00011) ([§4.4](#_Toc00113)). A base [interface](#_Trm00102) cannot be a type parameter on its own, though it can involve the type [parameters](#_Trm00059) that are in [scope](#_Trm00148). The following code illustrates how a class can implement and extend [constructed type](#_Trm00178)s:

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](#_Trm00430) of a [generic class declaration](#_Trm00299) must satisfy the uniqueness rule described in [§13.4.2](#_Toc00545).

### Explicit [interface](#_Trm00102) member implementations

For purposes of implementing [interface](#_Trm00102)s, a class or struct may declare ***explicit interface member implementations***. An [explicit](#_Trm00198) [interface](#_Trm00102) member implementation is a [method](#_Trm00056), [property](#_Trm00348), [event](#_Trm00088), or [indexer](#_Trm00087) declaration that [references](#_Trm00160) a fully qualified [interface](#_Trm00102) 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](#_Trm00198) [interface](#_Trm00102) member implementations.

In some cases, the name of an [interface](#_Trm00102) member may not be appropriate for the implementing class, in which case the [interface](#_Trm00102) member may be implemented using [explicit](#_Trm00198) [interface](#_Trm00102) member implementation. A class implementing a file [abstract](#_Trm00076)ion, for example, would likely implement a Close member function that has the effect of releasing the file [resource](#_Trm00296), and implement the Dispose [method](#_Trm00056) of the IDisposable [interface](#_Trm00102) using [explicit](#_Trm00198) [interface](#_Trm00102) 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](#_Trm00198) [interface](#_Trm00102) member implementation through its [fully qualified name](#_Trm00153) in a [method](#_Trm00056) invocation, [property](#_Trm00348) access, or [indexer](#_Trm00087) access. An [explicit](#_Trm00198) [interface](#_Trm00102) member implementation can only be accessed through an [interface](#_Trm00102) [instance](#_Trm00172), and is in that case referenced simply by its member name.

It is a compile-time error for an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation to include access modifiers, and it is a compile-time error to include the modifiers abstract, virtual, override, or static.

Explicit [interface](#_Trm00102) member implementations have different accessibility characteristics than other [members](#_Trm00012). Because [explicit](#_Trm00198) [interface](#_Trm00102) member implementations are never [accessible](#_Trm00138) through their [fully qualified name](#_Trm00153) in a [method](#_Trm00056) invocation or a [property](#_Trm00348) access, they are in a sense private. However, since they can be accessed through an [interface](#_Trm00102) [instance](#_Trm00172), they are in a sense also public.

Explicit [interface](#_Trm00102) member implementations serve two primary purposes:

* Because [explicit](#_Trm00198) [interface](#_Trm00102) member implementations are not [accessible](#_Trm00138) through class or struct [instance](#_Trm00172)s, they allow [interface](#_Trm00102) implementations to be excluded from the public [interface](#_Trm00102) of a class or struct. This is particularly useful when a class or struct implements an internal [interface](#_Trm00102) that is of no interest to a consumer of that class or struct.
* Explicit [interface](#_Trm00102) member implementations allow disambiguation of [interface](#_Trm00102) [members](#_Trm00012) with the same [signature](#_Trm00061). Without [explicit](#_Trm00198) [interface](#_Trm00102) member implementations it would be impossible for a class or struct to have different implementations of [interface](#_Trm00102) [members](#_Trm00012) with the same [signature](#_Trm00061) and [return type](#_Trm00060), as would it be impossible for a class or struct to have any implementation at all of [interface](#_Trm00102) [members](#_Trm00012) with the same [signature](#_Trm00061) but with different [return type](#_Trm00060)s.

For an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation to be valid, the class or struct must name an [interface](#_Trm00102) in its base class list that contains a member whose [fully qualified name](#_Trm00153), type, and parameter [types](#_Trm00011) exactly match those of the [explicit](#_Trm00198) [interface](#_Trm00102) 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](#_Trm00102) 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 [explicit](#_Trm00198)ly listed in the base class list of Ellipse.

The [fully qualified name](#_Trm00153) of an [interface](#_Trm00102) member must reference the [interface](#_Trm00102) 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](#_Trm00198) [interface](#_Trm00102) member implementation of Paint must be written as IControl.Paint.

### Uniqueness of implemented [interface](#_Trm00102)s

The [interface](#_Trm00102)s implemented by a generic type declaration must remain unique for all possible [constructed type](#_Trm00178)s. Without this rule, it would be impossible to determine the correct [method](#_Trm00056) to call for certain [constructed type](#_Trm00178)s. For example, suppose a [generic class declaration](#_Trm00299) 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](#_Trm00102) list of a generic type declaration is valid, the following steps are performed:

* Let L be the list of [interface](#_Trm00102)s directly specified in a generic class, struct, or [interface](#_Trm00102) declaration C.
* Add to L any [base interfaces](#_Trm00430) of the [interface](#_Trm00102)s already in L.
* Remove any duplicates from L.
* If any possible [constructed type](#_Trm00178) created from C would, [after](#_Trm00384) type [arguments](#_Trm00062) are substituted into L, cause two [interface](#_Trm00102)s in L to be identical, then the declaration of C is invalid. Constraint declarations are not considered when determining all possible [constructed type](#_Trm00178)s.

In the class declaration X above, the [interface](#_Trm00102) list L consists of I<U> and I<V>. The declaration is invalid because any [constructed type](#_Trm00178) with U and V being the same type would cause these two [interface](#_Trm00102)s to be identical [types](#_Trm00011).

It is possible for [interface](#_Trm00102)s specified at different [inheritance](#_Trm00047) 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](#_Trm00056) in Derived, since Derived<int,int> effectively re-implements I<int> ([§13.4.6](#_Toc00549)).

### Implementation of [generic method](#_Trm00333)s

When a [generic method](#_Trm00333) [implicit](#_Trm00197)ly implements an [interface](#_Trm00102) [method](#_Trm00056), the constraints given for each [method](#_Trm00056) type parameter must be equivalent in both declarations ([after](#_Trm00384) any [interface](#_Trm00102) type [parameters](#_Trm00059) are replaced with the appropriate type [arguments](#_Trm00062)), where [method](#_Trm00056) type [parameters](#_Trm00059) are identified by ordinal positions, left to right.

When a [generic method](#_Trm00333) [explicit](#_Trm00198)ly implements an [interface](#_Trm00102) [method](#_Trm00056), however, no constraints are allowed on the implementing [method](#_Trm00056). Instead, the constraints are [inherited](#_Trm00136) from the [interface](#_Trm00102) [method](#_Trm00056)

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](#_Trm00056) C.F<T> [implicit](#_Trm00197)ly 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](#_Trm00197) constraint on all type [parameters](#_Trm00059). The [method](#_Trm00056) C.G<T> [implicit](#_Trm00197)ly implements I<object,C,string>.G<T> because the constraints match those in the [interface](#_Trm00102), [after](#_Trm00384) the [interface](#_Trm00102) type [parameters](#_Trm00059) are replaced with the corresponding type [arguments](#_Trm00062). The constraint for [method](#_Trm00056) C.H<T> is an error because sealed [types](#_Trm00011) (string in this case) cannot be used as constraints. Omitting the constraint would also be an error since constraints of [implicit](#_Trm00197) [interface](#_Trm00102) [method](#_Trm00056) implementations are required to match. Thus, it is impossible to [implicit](#_Trm00197)ly implement I<object,C,string>.H<T>. This [interface](#_Trm00102) [method](#_Trm00056) can only be implemented using an [explicit](#_Trm00198) [interface](#_Trm00102) 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](#_Trm00198) [interface](#_Trm00102) member implementation invokes a public [method](#_Trm00056) having strictly weaker constraints. Note that the assignment from t to s is valid since T [inherits](#_Trm00315) 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](#_Trm00012) of the [interface](#_Trm00102)s that are listed in the base class list of the class or struct. The process of locating implementations of [interface](#_Trm00102) [members](#_Trm00012) 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](#_Trm00102) specified in the base class list of C. The implementation of a particular [interface](#_Trm00102) member I.M, where I is the [interface](#_Trm00102) 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](#_Trm00198) [interface](#_Trm00102) 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](#_Trm00178) where the two [members](#_Trm00012) as declared in the generic type have different [signature](#_Trm00061)s, but the type [arguments](#_Trm00062) make their [signature](#_Trm00061)s identical.

A compile-time error occurs if implementations cannot be located for all [members](#_Trm00012) of all [interface](#_Trm00102)s specified in the base class list of C. Note that the [members](#_Trm00012) of an [interface](#_Trm00102) include those [members](#_Trm00012) that are [inherited](#_Trm00136) from [base interfaces](#_Trm00430).

For purposes of [interface](#_Trm00102) mapping, a class member A matches an [interface](#_Trm00102) member B when:

* A and B are [method](#_Trm00056)s, 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](#_Trm00083) as B (A is permitted to have additional [accessors](#_Trm00083) if it is not an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation).
* A and B are [event](#_Trm00088)s, and the name and type of A and B are identical.
* A and B are [indexer](#_Trm00087)s, the type and formal parameter lists of A and B are identical, and A has the same [accessors](#_Trm00083) as B (A is permitted to have additional [accessors](#_Trm00083) if it is not an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation).

Notable implications of the [interface](#_Trm00102) mapping algorithm are:

* Explicit [interface](#_Trm00102) member implementations take [precedence](#_Trm00035) over other [members](#_Trm00012) in the same class or struct when determining the class or struct member that implements an [interface](#_Trm00102) member.
* Neither non-public nor static [members](#_Trm00012) participate in [interface](#_Trm00102) 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](#_Trm00198) [interface](#_Trm00102) member implementations take [precedence](#_Trm00035) over other [members](#_Trm00012).

If a class or struct implements two or more [interface](#_Trm00102)s containing a member with the same name, type, and parameter [types](#_Trm00011), it is possible to map each of those [interface](#_Trm00102) [members](#_Trm00012) 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 [method](#_Trm00056)s of both IControl and IForm are mapped onto the Paint [method](#_Trm00056) in Page. It is of course also possible to have separate [explicit](#_Trm00198) [interface](#_Trm00102) member implementations for the two [method](#_Trm00056)s.

If a class or struct implements an [interface](#_Trm00102) that contains [hidden](#_Trm00150) [members](#_Trm00012), then some [members](#_Trm00012) must necessarily be implemented through [explicit](#_Trm00198) [interface](#_Trm00102) member implementations. For example

interface IBase  
{  
 int P { get; }  
}  
  
interface IDerived: IBase  
{  
 new int P();  
}

An implementation of this [interface](#_Trm00102) would require at least one [explicit](#_Trm00198) [interface](#_Trm00102) 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 [interface](#_Trm00102)s that have the same base [interface](#_Trm00102), there can be only one implementation of the base [interface](#_Trm00102). 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](#_Trm00136) by ITextBox, and the IControl [inherited](#_Trm00136) by IListBox. Indeed, there is no notion of a separate identity for these [interface](#_Trm00102)s. Rather, the implementations of ITextBox and IListBox share the same implementation of IControl, and ComboBox is simply considered to implement three [interface](#_Trm00102)s, IControl, ITextBox, and IListBox.

The [members](#_Trm00012) of a base class participate in [interface](#_Trm00102) 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](#_Trm00056) F in Class1 is used in Class2's implementation of Interface1.

### Interface implementation [inheritance](#_Trm00047)

A class [inherits](#_Trm00315) all [interface](#_Trm00102) implementations provided by its [base classes](#_Trm00050).

Without [explicit](#_Trm00198)ly ***re-implementing*** an [interface](#_Trm00102), a derived class cannot in any way alter the [interface](#_Trm00102) mappings it [inherits](#_Trm00315) from its [base classes](#_Trm00050). 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](#_Trm00056) in TextBox [hide](#_Trm00132)s the Paint [method](#_Trm00056) in Control, but it does not alter the mapping of Control.Paint onto IControl.Paint, and calls to Paint through class [instance](#_Trm00172)s and [interface](#_Trm00102) [instance](#_Trm00172)s 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](#_Trm00102) [method](#_Trm00056) is mapped onto a virtual [method](#_Trm00056) in a class, it is possible for [derived classes](#_Trm00049) to override the virtual [method](#_Trm00056) and alter the implementation of the [interface](#_Trm00102). 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](#_Trm00198) [interface](#_Trm00102) member implementations cannot be declared virtual, it is not possible to override an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation. However, it is perfectly valid for an [explicit](#_Trm00198) [interface](#_Trm00102) member implementation to call another [method](#_Trm00056), and that other [method](#_Trm00056) can be declared virtual to allow [derived classes](#_Trm00049) 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](#_Trm00336) the PaintControl [method](#_Trm00056).

### Interface re-implementation

A class that [inherits](#_Trm00315) an [interface](#_Trm00102) implementation is permitted to ***re-implement*** the [interface](#_Trm00102) by including it in the base class list.

A [re-implement](#_Trm00435)ation of an [interface](#_Trm00102) follows exactly the same [interface](#_Trm00102) mapping rules as an initial implementation of an [interface](#_Trm00102). Thus, the [inherited](#_Trm00136) [interface](#_Trm00102) mapping has no effect whatsoever on the [interface](#_Trm00102) mapping established for the [re-implement](#_Trm00435)ation of the [interface](#_Trm00102). 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-implement](#_Trm00435)ation in MyControl, which maps IControl.Paint onto MyControl.Paint.

Inherited public member declarations and [inherited](#_Trm00136) [explicit](#_Trm00198) [interface](#_Trm00102) member declarations participate in the [interface](#_Trm00102) mapping process for [re-implement](#_Trm00435)ed [interface](#_Trm00102)s. 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](#_Trm00102) [method](#_Trm00056)s onto Derived.F, Base.IMethods.G, Derived.IMethods.H, and Base.I.

When a class implements an [interface](#_Trm00102), it [implicit](#_Trm00197)ly also implements all of that [interface](#_Trm00102)'s [base interfaces](#_Trm00430). Likewise, a [re-implement](#_Trm00435)ation of an [interface](#_Trm00102) is also [implicit](#_Trm00197)ly a [re-implement](#_Trm00435)ation of all of the [interface](#_Trm00102)'s [base interfaces](#_Trm00430). 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-implement](#_Trm00435)ation of IDerived also [re-implement](#_Trm00435)s IBase, mapping IBase.F onto D.F.

### Abstract classes and [interface](#_Trm00102)s

Like a non-[abstract](#_Trm00076) class, an [abstract](#_Trm00076) class must provide implementations of all [members](#_Trm00012) of the [interface](#_Trm00102)s that are listed in the base class list of the class. However, an [abstract](#_Trm00076) class is permitted to map [interface](#_Trm00102) [method](#_Trm00056)s onto [abstract](#_Trm00076) [method](#_Trm00056)s. 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](#_Trm00076) [method](#_Trm00056)s, which must be [overridden](#_Trm00075) in non-[abstract](#_Trm00076) classes that derive from C.

Note that [explicit](#_Trm00198) [interface](#_Trm00102) member implementations cannot be [abstract](#_Trm00076), but [explicit](#_Trm00198) [interface](#_Trm00102) member implementations are of course permitted to call [abstract](#_Trm00076) [method](#_Trm00056)s. 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](#_Trm00076) classes that derive from C would be required to override FF and GG, thus providing the actual implementation of IMethods.

# Enums

An ***enum type*** is a distinct [value](#_Trm00209) type ([§4.1](#_Toc00091)) that declares a set of named [constant](#_Trm00322)s.

The example

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

declares an [enum type](#_Trm00105) named Color with [members](#_Trm00012) Red, Green, and Blue.

## Enum declarations

An enum declaration declares a new [enum type](#_Trm00105). An enum declaration begins with the [keyword](#_Trm00117) enum, and defines the name, accessibility, [underlying type](#_Trm00106), and [members](#_Trm00012) 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](#_Trm00105) has a corresponding integral type called the ***underlying type*** of the [enum type](#_Trm00105). This [underlying type](#_Trm00106) must be able to represent all the enumerator [value](#_Trm00209)s [defined](#_Trm00121) in the enumeration. An enum declaration may [explicit](#_Trm00198)ly declare an [underlying type](#_Trm00106) of byte, sbyte, short, ushort, int, uint, long or ulong. Note that char cannot be used as an [underlying type](#_Trm00106). An enum declaration that does not [explicit](#_Trm00198)ly declare an [underlying type](#_Trm00106) has an [underlying type](#_Trm00106) of int.

The example

enum Color: long  
{  
 Red,  
 Green,  
 Blue  
}

declares an enum with an [underlying type](#_Trm00106) of long. A developer might choose to use an [underlying type](#_Trm00106) of long, as in the example, to enable the use of [value](#_Trm00209)s 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](#_Grm00143) may optionally include a [sequence](#_Trm00259) 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 ([§10.1.1](#_Toc00390)). Note, however, that the abstract and sealed modifiers are not permitted in an enum declaration. Enums cannot be [abstract](#_Trm00076) and do not permit derivation.

## Enum [members](#_Trm00012)

The body of an [enum type](#_Trm00105) declaration defines zero or more enum [members](#_Trm00012), which are the named [constant](#_Trm00322)s of the [enum type](#_Trm00105). No two enum [members](#_Trm00012) 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](#_Trm00322) [value](#_Trm00209). The type of this [value](#_Trm00209) is the [underlying type](#_Trm00106) for the containing enum. The [constant](#_Trm00322) [value](#_Trm00209) for each enum member must be in the range of the [underlying type](#_Trm00106) for the enum. The example

enum Color: uint  
{  
 Red = -1,  
 Green = -2,  
 Blue = -3  
}

results in a compile-time error because the [constant](#_Trm00322) [value](#_Trm00209)s -1, -2, and -3 are not in the range of the underlying integral type uint.

Multiple enum [members](#_Trm00012) may share the same associated [value](#_Trm00209). The example

enum Color  
{  
 Red,  
 Green,  
 Blue,  
  
 Max = Blue  
}

shows an enum in which two enum [members](#_Trm00012) -- Blue and Max -- have the same associated [value](#_Trm00209).

The associated [value](#_Trm00209) of an enum member is assigned either [implicit](#_Trm00197)ly or [explicit](#_Trm00198)ly. If the declaration of the enum member has a [constant\_expression](#_Grm00068) initializer, the [value](#_Trm00209) of that [constant](#_Trm00322) expression, [implicit](#_Trm00197)ly converted to the [underlying type](#_Trm00106) of the enum, is the associated [value](#_Trm00209) of the enum member. If the declaration of the enum member has no initializer, its associated [value](#_Trm00209) is set [implicit](#_Trm00197)ly, as follows:

* If the enum member is the first enum member declared in the [enum type](#_Trm00105), its associated [value](#_Trm00209) is zero.
* Otherwise, the associated [value](#_Trm00209) of the enum member is obtained by increasing the associated [value](#_Trm00209) of the textually preceding enum member by one. This increased [value](#_Trm00209) must be within the range of [value](#_Trm00209)s that can be represented by the [underlying type](#_Trm00106), 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 [value](#_Trm00209)s. The output is:

Red = 0  
Green = 10  
Blue = 11

for the following reasons:

* the enum member Red is automatically assigned the [value](#_Trm00209) zero (since it has no initializer and is the first enum member);
* the enum member Green is [explicit](#_Trm00198)ly given the [value](#_Trm00209) 10;
* and the enum member Blue is automatically assigned the [value](#_Trm00209) one greater than the member that textually precedes it.

The associated [value](#_Trm00209) of an enum member may not, directly or indirectly, use the [value](#_Trm00209) 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, [value](#_Trm00209)s of other enum [members](#_Trm00012) are always treated as having the type of their [underlying type](#_Trm00106), so that casts are not necessary when referring to other enum [members](#_Trm00012).

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](#_Trm00306) B [explicit](#_Trm00198)ly, and B [depends on](#_Trm00306) A [implicit](#_Trm00197)ly.

Enum [members](#_Trm00012) are named and [scope](#_Trm00148)d in a manner exactly analogous to [field](#_Trm00323)s within classes. The [scope](#_Trm00148) of an enum member is the body of its containing [enum type](#_Trm00105). Within that [scope](#_Trm00148), enum [members](#_Trm00012) 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](#_Trm00105). Enum [members](#_Trm00012) do not have any [declared accessibility](#_Trm00140) -- an enum member is [accessible](#_Trm00138) if its containing [enum type](#_Trm00105) is [accessible](#_Trm00138).

## The System.Enum type

The type System.Enum is the [abstract](#_Trm00076) base class of all [enum type](#_Trm00105)s (this is distinct and different from the [underlying type](#_Trm00106) of the [enum type](#_Trm00105)), and the [members](#_Trm00012) [inherited](#_Trm00136) from System.Enum are available in any [enum type](#_Trm00105). A [boxing](#_Trm00029) [conversion](#_Trm00196) ([§4.3.1](#_Toc00111)) exists from any [enum type](#_Trm00105) to System.Enum, and an un[boxing](#_Trm00029) [conversion](#_Trm00196) ([§4.3.2](#_Toc00112)) exists from System.Enum to any [enum type](#_Trm00105).

Note that System.Enum is not itself an [enum\_type](#_Grm00029). Rather, it is a [class\_type](#_Grm00030) from which all [enum\_type](#_Grm00029)s are derived. The type System.Enum [inherits](#_Trm00315) from the type System.ValueType ([§4.1.1](#_Toc00092)), which, in turn, [inherits](#_Trm00315) from type object. At run-time, a [value](#_Trm00209) of type System.Enum can be null or a reference to a boxed [value](#_Trm00209) of any [enum type](#_Trm00105).

## Enum [value](#_Trm00209)s and operations

Each [enum type](#_Trm00105) defines a distinct type; an [explicit](#_Trm00198) enumeration [conversion](#_Trm00196) ([§6.2.2](#_Toc00183)) is required to convert between an [enum type](#_Trm00105) and an integral type, or between two [enum type](#_Trm00105)s. The set of [value](#_Trm00209)s that an [enum type](#_Trm00105) can take on is not limited by its enum [members](#_Trm00012). In particular, any [value](#_Trm00209) of the [underlying type](#_Trm00106) of an enum can be cast to the [enum type](#_Trm00105), and is a distinct valid [value](#_Trm00209) of that [enum type](#_Trm00105).

Enum [members](#_Trm00012) have the type of their containing [enum type](#_Trm00105) (except within other enum member initializers: see [§14.3](#_Toc00554)). The [value](#_Trm00209) of an enum member declared in [enum type](#_Trm00105) E with associated [value](#_Trm00209) v is (E)v.

The following [operator](#_Trm00090)s can be used on [value](#_Trm00209)s of [enum type](#_Trm00105)s: ==, !=, <, >, <=, >= ([§7.10.5](#_Toc00304)), binary + ([§7.8.4](#_Toc00296)), binary - ([§7.8.5](#_Toc00297)), ^, &, | ([§7.11.2](#_Toc00313)), ~ ([§7.7.4](#_Toc00285)), ++ and -- ([§7.6.9](#_Toc00269) and [§7.7.5](#_Toc00286)).

Every [enum type](#_Trm00105) automatically derives from the class System.Enum (which, in turn, derives from System.ValueType and object). Thus, [inherited](#_Trm00136) [method](#_Trm00056)s and properties of this class can be used on [value](#_Trm00209)s of an [enum type](#_Trm00105).

# Delegates

Delegates enable scenarios that other languages—such as C++, Pascal, and Modula -- have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully [object](#_Trm00173) oriented, and unlike C++ pointers to member functions, delegates encapsulate both an [object](#_Trm00173) [instance](#_Trm00172) and a [method](#_Trm00056).

A delegate declaration defines a class that is derived from the class System.Delegate. A delegate [instance](#_Trm00172) encapsulates an invocation list, which is a list of one or more [method](#_Trm00056)s, each of which is referred to as a callable entity. For [instance](#_Trm00172) [method](#_Trm00056)s, a callable entity consists of an [instance](#_Trm00172) and a [method](#_Trm00056) on that [instance](#_Trm00172). For static [method](#_Trm00056)s, a callable entity consists of just a [method](#_Trm00056). Invoking a delegate [instance](#_Trm00172) with an appropriate set of [arguments](#_Trm00062) causes each of the delegate's callable entities to be invoked with the given set of [arguments](#_Trm00062).

An interesting and useful [property](#_Trm00348) of a delegate [instance](#_Trm00172) is that it does not know or care about the classes of the [method](#_Trm00056)s it encapsulates; all that matters is that those [method](#_Trm00056)s be compatible ([§15.1](#_Toc00558)) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

## Delegate declarations

A [delegate\_declaration](#_Grm00146) is a [type\_declaration](#_Grm00105) ([§9.6](#_Toc00385)) that declares a new [delegate type](#_Trm00107).

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 [hide](#_Trm00132)s an [inherited](#_Trm00136) member by the same name, as described in [§10.3.4](#_Toc00414).

The public, protected, internal, and private modifiers control the accessibility of the [delegate type](#_Trm00107). Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted ([§3.5.1](#_Toc00077)).

The delegate's type name is [identifier](#_Grm00007).

The optional [formal\_parameter\_list](#_Grm00117) specifies the [parameters](#_Trm00059) of the delegate, and [return\_type](#_Grm00116) indicates the [return type](#_Trm00060) of the delegate.

The optional [variant\_type\_parameter\_list](#_Grm00135) ([§13.1.3](#_Toc00531)) specifies the type [parameters](#_Trm00059) to the delegate itself.

The [return type](#_Trm00060) of a [delegate type](#_Trm00107) must be either void, or [output-safe](#_Trm00427) ([§13.1.3.1](#_Toc00532)).

All the formal parameter [types](#_Trm00011) of a [delegate type](#_Trm00107) must be [input-safe](#_Trm00428). Additionally, any out or ref parameter [types](#_Trm00011) must also be [output-safe](#_Trm00427). Note that even out [parameters](#_Trm00059) are required to be [input-safe](#_Trm00428), due to a limitiation of the underlying execution platform.

Delegate [types](#_Trm00011) in C# are name equivalent, not structurally equivalent. Specifically, two different [delegate type](#_Trm00107)s that have the same parameter lists and [return type](#_Trm00060) are considered different [delegate type](#_Trm00107)s. However, [instance](#_Trm00172)s of two distinct but structurally equivalent [delegate type](#_Trm00107)s may compare as equal ([§7.10.8](#_Toc00307)).

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 [method](#_Trm00056)s A.M1 and B.M1are compatible with both the [delegate type](#_Trm00107)s D1 and D2 , since they have the same [return type](#_Trm00060) and parameter list; however, these [delegate type](#_Trm00107)s are two different [types](#_Trm00011), so they are not interchangeable. The [method](#_Trm00056)s B.M2, B.M3, and B.M4 are incompatible with the [delegate type](#_Trm00107)s D1 and D2, since they have different [return type](#_Trm00060)s or parameter lists.

Like other generic [type declarations](#_Trm00028), type [arguments](#_Trm00062) must be given to create a constructed [delegate type](#_Trm00107). The parameter [types](#_Trm00011) and [return type](#_Trm00060) of a constructed [delegate type](#_Trm00107) are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed [delegate type](#_Trm00107). The resulting [return type](#_Trm00060) and parameter [types](#_Trm00011) are used in determining what [method](#_Trm00056)s are compatible with a constructed [delegate type](#_Trm00107). For example:

delegate bool Predicate<T>(T value);  
  
class X  
{  
 static bool F(int i) {...}  
 static bool G(string s) {...}  
}

The [method](#_Trm00056) X.F is compatible with the [delegate type](#_Trm00107) Predicate<int> and the [method](#_Trm00056) X.G is compatible with the [delegate type](#_Trm00107) Predicate<string> .

The only way to declare a [delegate type](#_Trm00107) is via a [delegate\_declaration](#_Grm00146). A [delegate type](#_Trm00107) is a class type that is derived from System.Delegate. Delegate [types](#_Trm00011) are [implicit](#_Trm00197)ly sealed, so it is not permissible to derive any type from a [delegate type](#_Trm00107). 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](#_Trm00107); it is a class type from which all [delegate type](#_Trm00107)s 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](#_Trm00172) can also be applied to a delegate class or [instance](#_Trm00172), respectively. In particular, it is possible to access [members](#_Trm00012) of the System.Delegate type via the usual member access syntax.

The set of [method](#_Trm00056)s encapsulated by a delegate [instance](#_Trm00172) is called an invocation list. When a delegate [instance](#_Trm00172) is created ([§15.2](#_Toc00559)) from a single [method](#_Trm00056), it encapsulates that [method](#_Trm00056), and its invocation list contains only one entry. However, when two non-null delegate [instance](#_Trm00172)s are combined, their invocation lists are concatenated -- in the order left operand then right operand -- to form a new invocation list, which contains two or more entries.

Delegates are combined using the binary + ([§7.8.4](#_Toc00296)) and += [operator](#_Trm00090)s ([§7.17.2](#_Toc00343)). A delegate can be removed from a combination of delegates, using the binary - ([§7.8.5](#_Toc00297)) and -= [operator](#_Trm00090)s ([§7.17.2](#_Toc00343)). Delegates can be compared for equality ([§7.10.8](#_Toc00307)).

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](#_Trm00256), they each encapsulate one [method](#_Trm00056). When cd3 is [instantiated](#_Trm00256), it has an invocation list of two [method](#_Trm00056)s, M1 and M2, in that order. cd4's invocation list contains M1, M2, and M1, in that order. Finally, cd5's invocation list contains M1, M2, M1, M1, and M2, in that order. For more examples of combining (as well as removing) delegates, see [§15.4](#_Toc00561).

## Delegate compatibility

A [method](#_Trm00056) or delegate M is ***compatible*** with a [delegate type](#_Trm00107) D if all of the following are true:

* D and M have the same number of [parameters](#_Trm00059), and each parameter in D has the same ref or out modifiers as the corresponding parameter in M.
* For each [value](#_Trm00209) parameter (a parameter with no ref or out modifier), an identity [conversion](#_Trm00196) ([§6.1.1](#_Toc00169)) or [implicit](#_Trm00197) reference [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) 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](#_Trm00197) reference [conversion](#_Trm00196) exists from the [return type](#_Trm00060) of M to the [return type](#_Trm00060) of D.

## Delegate instantiation

An [instance](#_Trm00172) of a delegate is created by a [delegate\_creation\_expression](#_Grm00048) ([§7.6.10.5](#_Toc00275)) or a [conversion](#_Trm00196) to a [delegate type](#_Trm00107). The newly created delegate [instance](#_Trm00172) then refers to either:

* The static [method](#_Trm00056) referenced in the [delegate\_creation\_expression](#_Grm00048), or
* The [target](#_Trm00290) [object](#_Trm00173) (which cannot be null) and [instance](#_Trm00172) [method](#_Trm00056) referenced in the [delegate\_creation\_expression](#_Grm00048), 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](#_Trm00256), delegate [instance](#_Trm00172)s always refer to the same [target](#_Trm00290) [object](#_Trm00173) and [method](#_Trm00056). 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](#_Trm00172) whose invocation list contains one entry is invoked, it invokes the one [method](#_Trm00056) with the same [arguments](#_Trm00062) it was given, and returns the same [value](#_Trm00209) as the referred to [method](#_Trm00056). (See [§7.6.5.3](#_Toc00263) 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](#_Trm00056) that was invoked, the search for an exception catch clause continues in the [method](#_Trm00056) that called the delegate, as if that [method](#_Trm00056) had directly called the [method](#_Trm00056) to which that delegate referred.

Invocation of a delegate [instance](#_Trm00172) whose invocation list contains multiple entries proceeds by invoking each of the [method](#_Trm00056)s in the invocation list, synchronously, in order. Each [method](#_Trm00056) so called is passed the same set of [arguments](#_Trm00062) as was given to the delegate [instance](#_Trm00172). If such a delegate invocation includes reference [parameters](#_Trm00059) ([§10.6.1.2](#_Toc00444)), each [method](#_Trm00056) invocation will occur with a reference to the same variable; changes to that variable by one [method](#_Trm00056) in the invocation list will be [visible](#_Trm00152) to [method](#_Trm00056)s further down the invocation list. If the delegate invocation includes [output parameter](#_Trm00065)s or a return [value](#_Trm00209), their final [value](#_Trm00209) 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](#_Trm00056) that was invoked, the search for an exception catch clause continues in the [method](#_Trm00056) that called the delegate, and any [method](#_Trm00056)s further down the invocation list are not invoked.

Attempting to invoke a delegate [instance](#_Trm00172) whose [value](#_Trm00209) 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

Exceptions in C# provide a structured, uniform, and [type-safe](#_Trm00006) way of handling both system level and [application](#_Trm00124) 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](#_Trm00172) of a class type derived from System.Exception. In C++, any [value](#_Trm00209) of any type can be used to represent an exception.
* In C#, a finally [block](#_Trm00038) ([§8.10](#_Toc00372)) 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 de[references](#_Trm00160) have well [defined](#_Trm00121) exception classes and are on a par with [application](#_Trm00124)-level error conditions.

## Causes of exceptions

Exception can be thrown in two different ways.

* A throw statement ([§8.9.5](#_Toc00371)) 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](#_Trm00037) and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation ([§7.8.2](#_Toc00294)) throws a System.DivideByZeroException if the denominator is zero. See [§16.4](#_Toc00566) 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](#_Trm00355) [property](#_Trm00348) of type string that contains a human-readable description of the reason for the exception.
* InnerException is a [read-only](#_Trm00355) [property](#_Trm00348) of type Exception. If its [value](#_Trm00209) is non-null, it refers to the exception that caused the current exception—that is, the current exception was raised in a catch [block](#_Trm00038) handling the InnerException. Otherwise, its [value](#_Trm00209) is null, indicating that this exception was not caused by another exception. The number of exception [object](#_Trm00173)s chained together in this manner can be arbitrary.

The [value](#_Trm00209) of these properties can be specified in calls to the [instance](#_Trm00172) constructor for System.Exception.

## How exceptions are handled

Exceptions are handled by a try statement ([§8.10](#_Toc00372)).

When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the [run-time type](#_Trm00073) of the exception. First, the current [method](#_Trm00056) 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](#_Trm00056) that called the current [method](#_Trm00056) is searched for a lexically enclosing try statement that encloses the point of the call to the current [method](#_Trm00056). 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](#_Trm00073) 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](#_Trm00037) more [nested](#_Trm00143) 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](#_Trm00081) ([§10.12](#_Toc00481)) or static [field](#_Trm00323) initializer, then a System.TypeInitializationException is thrown at the point that triggered the invocation of the [static constructor](#_Trm00081). 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](#_Trm00121).

Exceptions that occur during [destructor](#_Trm00091) execution are worth special mention. If an exception occurs during [destructor](#_Trm00091) execution, and that exception is not caught, then the execution of that [destructor](#_Trm00091) is terminated and the [destructor](#_Trm00091) 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](#_Trm00091), then the exception is discarded.

## Common Exception [Classes](#_Trm00044)

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](#_Trm00093) fails because the actual type of the stored element is in[compatible](#_Trm00438) with the actual type of the [array](#_Trm00093). |
| System.DivideByZeroException | Thrown when an attempt to divide an integral [value](#_Trm00209) by zero occurs. |
| System.IndexOutOfRangeException | Thrown when an attempt to index an [array](#_Trm00093) via an index that is less than zero or outside the bounds of the [array](#_Trm00093). |
| System.InvalidCastException | Thrown when an [explicit](#_Trm00198) [conversion](#_Trm00196) from a base type or [interface](#_Trm00102) 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](#_Trm00173) 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](#_Trm00056) calls; typically indicative of very deep or unbounded recursion. |
| System.TypeInitializationException | Thrown when a [static constructor](#_Trm00081) throws an exception, and no catch clauses exists to catch it. |

# Attributes

Much of the C# language enables the [program](#_Trm00109)mer to specify declarative information about the entities [defined](#_Trm00121) in the [program](#_Trm00109). For example, the accessibility of a [method](#_Trm00056) in a class is specified by decorating it with the [method\_modifier](#_Grm00116)s public, protected, internal, and private.

C# enables [program](#_Trm00109)mers to invent new kinds of declarative information, called ***attributes***. Programmers can then attach [attributes](#_Trm00108) to various [program](#_Trm00109) entities, and retrieve attribute information in a run-time environment. For [instance](#_Trm00172), a framework might define a HelpAttribute attribute that can be placed on certain [program](#_Trm00109) [elements](#_Trm00094) (such as classes and [method](#_Trm00056)s) to provide a mapping from those [program](#_Trm00109) [elements](#_Trm00094) to their documentation.

Attributes are [defined](#_Trm00121) through the declaration of attribute classes ([§17.1](#_Toc00568)), which may have positional and named [parameters](#_Trm00059) ([§17.1.2](#_Toc00570)). Attributes are attached to entities in a C# [program](#_Trm00109) using attribute specifications ([§17.2](#_Toc00572)), and can be retrieved at run-time as attribute [instance](#_Trm00172)s ([§17.3](#_Toc00573)).

## Attribute classes

A class that derives from the [abstract](#_Trm00076) class System.Attribute, whether directly or indirectly, is an ***attribute class***. The declaration of an [attribute class](#_Trm00440) defines a new kind of ***attribute*** that can be placed on a declaration. By convention, [attribute](#_Trm00441) classes are named with a suffix of Attribute. Uses of an [attribute](#_Trm00441) may either include or omit this suffix.

### Attribute usage

The [attribute](#_Trm00441) AttributeUsage ([§17.4.1](#_Toc00577)) is used to describe how an [attribute](#_Trm00441) class can be used.

AttributeUsage has a positional parameter ([§17.1.2](#_Toc00570)) that enables an [attribute](#_Trm00441) 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](#_Trm00441) class named SimpleAttribute that can be placed on [class\_declaration](#_Grm00107)s and [interface\_declaration](#_Grm00133)s only. The example

[Simple] class Class1 {...}  
  
[Simple] interface Interface1 {...}

shows several uses of the Simple [attribute](#_Trm00441). Although this [attribute](#_Trm00441) is [defined](#_Trm00121) with the name SimpleAttribute, when this [attribute](#_Trm00441) is used, the Attribute suffix may be omitted, resulting in the short name Simple. Thus, the example above is semantically equivalent to the following:

[SimpleAttribute] class Class1 {...}  
  
[SimpleAttribute] interface Interface1 {...}

AttributeUsage has a named parameter ([§17.1.2](#_Toc00570)) called AllowMultiple, which indicates whether the [attribute](#_Trm00441) can be specified more than once for a given entity. If AllowMultiple for an [attribute](#_Trm00441) class is true, then that [attribute](#_Trm00441) class is a ***multi-use attribute class***, and can be specified more than once on an entity. If AllowMultiple for an [attribute](#_Trm00441) class is false or it is unspecified, then that [attribute](#_Trm00441) 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](#_Trm00441) class named AuthorAttribute. The example

[Author("Brian Kernighan"), Author("Dennis Ritchie")]  
class Class1  
{  
 ...  
}

shows a class declaration with two uses of the Author [attribute](#_Trm00441).

AttributeUsage has another named parameter called Inherited, which indicates whether the [attribute](#_Trm00441), when specified on a base class, is also [inherited](#_Trm00136) by classes that derive from that base class. If Inherited for an [attribute](#_Trm00441) class is true, then that [attribute](#_Trm00441) is [inherited](#_Trm00136). If Inherited for an [attribute](#_Trm00441) class is false then that [attribute](#_Trm00441) is not [inherited](#_Trm00136). If it is unspecified, its [default value](#_Trm00164) is true.

An [attribute](#_Trm00441) class X not having an AttributeUsage [attribute](#_Trm00441) 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](#_Trm00059)

Attribute classes can have ***positional parameters*** and ***named parameters***. Each public [instance](#_Trm00172) constructor for an [attribute](#_Trm00441) class defines a valid [sequence](#_Trm00259) of positional [parameters](#_Trm00059) for that [attribute](#_Trm00441) class. Each non-static public [read-write](#_Trm00354) [field](#_Trm00323) and [property](#_Trm00348) for an [attribute](#_Trm00441) class defines a named parameter for the [attribute](#_Trm00441) 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](#_Trm00441) class named HelpAttribute that has one positional parameter, url, and one named parameter, Topic. Although it is non-static and public, the [property](#_Trm00348) Url does not define a named parameter, since it is not [read-write](#_Trm00354).

This [attribute](#_Trm00441) 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](#_Trm00011)

The [types](#_Trm00011) of positional and [named parameters](#_Trm00445) for an [attribute](#_Trm00441) class are limited to the ***attribute parameter types***, which are:

* One of the following [types](#_Trm00011): bool, byte, char, double, float, int, long, sbyte, short, string, uint, ulong, ushort.
* The type object.
* The type System.Type.
* An [enum type](#_Trm00105), provided it has public accessibility and the [types](#_Trm00011) in which it is [nested](#_Trm00143) (if any) also have public accessibility ([§17.2](#_Toc00572)).
* Single-dimensional [array](#_Trm00093)s of the above [types](#_Trm00011).
* A constructor argument or public [field](#_Trm00323) which does not have one of these [types](#_Trm00011), cannot be used as a positional or named parameter in an [attribute](#_Trm00441) specification.

## Attribute specification

***Attribute specification*** is the [application](#_Trm00124) of a previously [defined](#_Trm00121) [attribute](#_Trm00441) to a declaration. An [attribute](#_Trm00441) is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global [scope](#_Trm00148) (to specify [attribute](#_Trm00441)s on the containing assembly or module) and for [type\_declaration](#_Grm00105)s ([§9.6](#_Toc00385)), [class\_member\_declaration](#_Grm00113)s ([§10.1.5](#_Toc00399)), [interface\_member\_declaration](#_Grm00138)s ([§13.2](#_Toc00536)), [struct\_member\_declaration](#_Grm00130)s ([§11.2](#_Toc00503)), [enum\_member\_declaration](#_Grm00145)s ([§14.3](#_Toc00554)), [accessor\_declarations](#_Grm00119) ([§10.7.2](#_Toc00459)), [event\_accessor\_declarations](#_Grm00120) ([§10.8.1](#_Toc00464)), and [formal\_parameter\_list](#_Grm00117)s ([§10.6.1](#_Toc00442)).

Attributes are specified in ***attribute sections***. An [attribute](#_Trm00441) section consists of a pair of square brackets, which surround a comma-separated list of one or more [attribute](#_Trm00441)s. The order in which [attribute](#_Trm00441)s are specified in such a list, and the order in which sections attached to the same [program](#_Trm00109) entity are arranged, is not significant. For [instance](#_Trm00172), the [attribute](#_Trm00441) 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](#_Trm00441) consists of an [attribute\_name](#_Grm00147) and an optional list of positional and named [arguments](#_Trm00062). The positional [arguments](#_Trm00062) (if any) precede the named [arguments](#_Trm00062). A [positional argument](#_Trm00230) consists of an [attribute\_argument\_expression](#_Grm00147); a [named argument](#_Trm00229) consists of a name, followed by an equal sign, followed by an [attribute\_argument\_expression](#_Grm00147), which, together, are constrained by the same rules as simple assignment. The order of named [arguments](#_Trm00062) is not significant.

The [attribute\_name](#_Grm00147) identifies an [attribute](#_Trm00441) class. If the form of [attribute\_name](#_Grm00147) is [type\_name](#_Grm00027) then this name must refer to an [attribute](#_Trm00441) 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](#_Trm00441) class when Class1 is not an [attribute](#_Trm00441) class.

Certain contexts permit the specification of an [attribute](#_Trm00441) on more than one [target](#_Trm00290). A [program](#_Trm00109) can [explicit](#_Trm00198)ly specify the [target](#_Trm00290) by including an [attribute\_target\_specifier](#_Grm00147). When an [attribute](#_Trm00441) is placed at the global level, a [global\_attribute\_target\_specifier](#_Grm00147) is required. In all other locations, a reasonable default is applied, but an [attribute\_target\_specifier](#_Grm00147) 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](#_Grm00147)s can be omitted except at the global level. The potentially ambiguous contexts are resolved as follows:

* An [attribute](#_Trm00441) specified at global [scope](#_Trm00148) can apply either to the [target](#_Trm00290) assembly or the [target](#_Trm00290) module. No default exists for this context, so an [attribute\_target\_specifier](#_Grm00147) is always required in this context. The presence of the assembly [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [target](#_Trm00290) assembly; the presence of the module [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [target](#_Trm00290) module.
* An [attribute](#_Trm00441) specified on a delegate declaration can apply either to the delegate being declared or to its return [value](#_Trm00209). In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the delegate. The presence of the type [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the delegate; the presence of the return [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the return [value](#_Trm00209).
* An [attribute](#_Trm00441) specified on a [method](#_Trm00056) declaration can apply either to the [method](#_Trm00056) being declared or to its return [value](#_Trm00209). In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the [method](#_Trm00056). The presence of the method [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [method](#_Trm00056); the presence of the return [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the return [value](#_Trm00209).
* An [attribute](#_Trm00441) specified on an [operator](#_Trm00090) declaration can apply either to the [operator](#_Trm00090) being declared or to its return [value](#_Trm00209). In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the [operator](#_Trm00090). The presence of the method [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [operator](#_Trm00090); the presence of the return [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the return [value](#_Trm00209).
* An [attribute](#_Trm00441) specified on an [event](#_Trm00088) declaration that omits [event](#_Trm00088) [accessors](#_Trm00083) can apply to the [event](#_Trm00088) being declared, to the associated [field](#_Trm00323) (if the [event](#_Trm00088) is not [abstract](#_Trm00076)), or to the associated add and remove [method](#_Trm00056)s. In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the [event](#_Trm00088). The presence of the event [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [event](#_Trm00088); the presence of the field [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [field](#_Trm00323); and the presence of the method [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [method](#_Trm00056)s.
* An [attribute](#_Trm00441) specified on a get accessor declaration for a [property](#_Trm00348) or [indexer](#_Trm00087) declaration can apply either to the associated [method](#_Trm00056) or to its return [value](#_Trm00209). In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the [method](#_Trm00056). The presence of the method [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [method](#_Trm00056); the presence of the return [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the return [value](#_Trm00209).
* An [attribute](#_Trm00441) specified on a set accessor for a [property](#_Trm00348) or [indexer](#_Trm00087) declaration can apply either to the associated [method](#_Trm00056) or to its lone [implicit](#_Trm00197) parameter. In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the [method](#_Trm00056). The presence of the method [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [method](#_Trm00056); the presence of the param [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the parameter; the presence of the return [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the return [value](#_Trm00209).
* An [attribute](#_Trm00441) specified on an add or remove accessor declaration for an [event](#_Trm00088) declaration can apply either to the associated [method](#_Trm00056) or to its lone parameter. In the absence of an [attribute\_target\_specifier](#_Grm00147), the [attribute](#_Trm00441) applies to the [method](#_Trm00056). The presence of the method [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the [method](#_Trm00056); the presence of the param [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the parameter; the presence of the return [attribute\_target\_specifier](#_Grm00147) indicates that the [attribute](#_Trm00441) applies to the return [value](#_Trm00209).

In other contexts, inclusion of an [attribute\_target\_specifier](#_Grm00147) is permitted but unnecessary. For [instance](#_Trm00172), 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](#_Grm00147). For [instance](#_Trm00172), the specifier param cannot be used on a class declaration:

[param: Author("Brian Kernighan")] // Error  
class Class1 {}

By convention, [attribute](#_Trm00441) classes are named with a suffix of Attribute. An [attribute\_name](#_Grm00147) of the form [type\_name](#_Grm00027) may either include or omit this suffix. If an [attribute](#_Trm00441) class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the [attribute\_name](#_Grm00147) is spelled such that its right-most [identifier](#_Grm00007) is a [verbatim identifier](#_Trm00116) ([§2.4.2](#_Toc00044)), then only an [attribute](#_Trm00441) 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](#_Trm00441) classes named X and XAttribute. The [attribute](#_Trm00441) [X] is ambiguous, since it could refer to either X or XAttribute. Using a [verbatim identifier](#_Trm00116) allows the exact intent to be specified in such rare cases. The [attribute](#_Trm00441) [XAttribute] is not ambiguous (although it would be if there was an [attribute](#_Trm00441) class named XAttributeAttribute!). If the declaration for class X is removed, then both [attribute](#_Trm00441)s refer to the [attribute](#_Trm00441) 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](#_Trm00441) 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](#_Trm00441) class, more than once on the declaration of Class1.

An expression E is an [attribute\_argument\_expression](#_Grm00147) if all of the following [statements](#_Trm00037) are true:

* The type of E is an [attribute](#_Trm00441) parameter type ([§17.1.3](#_Toc00571)).
* At compile-time, the [value](#_Trm00209) of E can be resolved to one of the following:
  + A [constant](#_Trm00322) [value](#_Trm00209).
  + A System.Type [object](#_Trm00173).
  + A one-dimensional [array](#_Trm00093) of [attribute\_argument\_expression](#_Grm00147)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](#_Grm00050) ([§7.6.11](#_Toc00277)) used as an [attribute](#_Trm00441) argument expression can reference a non-generic type, a closed [constructed type](#_Trm00178), or an [unbound generic type](#_Trm00176), 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 [instance](#_Trm00172)s

An ***attribute instance*** is an [instance](#_Trm00172) that represents an [attribute](#_Trm00441) at run-time. An [attribute](#_Trm00441) is [defined](#_Trm00121) with an [attribute](#_Trm00441) class, positional [arguments](#_Trm00062), and named [arguments](#_Trm00062). An [attribute](#_Trm00441) [instance](#_Trm00172) is an [instance](#_Trm00172) of the [attribute](#_Trm00441) class that is initialized with the positional and named [arguments](#_Trm00062).

Retrieval of an [attribute](#_Trm00441) [instance](#_Trm00172) involves both compile-time and run-time processing, as described in the following sections.

### Compilation of an [attribute](#_Trm00441)

The compilation of an [attribute](#_Grm00147) with [attribute](#_Trm00441) class T, [positional\_argument\_list](#_Grm00147) P and [named\_argument\_list](#_Grm00147) N, consists of the following steps:

* Follow the compile-time processing steps for compiling an [object\_creation\_expression](#_Grm00044) of the form new T(P). These steps either result in a compile-time error, or determine an [instance](#_Trm00172) 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](#_Grm00147) Arg in N:
  + Let Name be the [identifier](#_Grm00007) of the [named\_argument](#_Grm00147) Arg.
  + Name must identify a non-static [read-write](#_Trm00354) public [field](#_Trm00323) or [property](#_Trm00348) on T. If T has no such [field](#_Trm00323) or [property](#_Trm00348), then a compile-time error occurs.
* Keep the following information for run-time instantiation of the [attribute](#_Trm00441): the [attribute](#_Trm00441) class T, the [instance](#_Trm00172) constructor C on T, the [positional\_argument\_list](#_Grm00147) P and the [named\_argument\_list](#_Grm00147) N.

### Run-time retrieval of an [attribute](#_Trm00441) [instance](#_Trm00172)

Compilation of an [attribute](#_Grm00147) yields an [attribute](#_Trm00441) class T, an [instance](#_Trm00172) constructor C on T, a [positional\_argument\_list](#_Grm00147) P, and a [named\_argument\_list](#_Grm00147) N. Given this information, an [attribute](#_Trm00441) [instance](#_Trm00172) can be retrieved at run-time using the following steps:

* Follow the run-time processing steps for executing an [object\_creation\_expression](#_Grm00044) of the form new T(P), using the [instance](#_Trm00172) constructor C as determined at compile-time. These steps either result in an exception, or produce an [instance](#_Trm00172) O of T.
* For each [named\_argument](#_Grm00147) Arg in N, in order:
  + Let Name be the [identifier](#_Grm00007) of the [named\_argument](#_Grm00147) Arg. If Name does not identify a non-static public [read-write](#_Trm00354) [field](#_Trm00323) or [property](#_Trm00348) on O, then an exception is thrown.
  + Let Value be the result of evaluating the [attribute\_argument\_expression](#_Grm00147) of Arg.
  + If Name identifies a [field](#_Trm00323) on O, then set this [field](#_Trm00323) to Value.
  + Otherwise, Name identifies a [property](#_Trm00348) on O. Set this [property](#_Trm00348) to Value.
  + The result is O, an [instance](#_Trm00172) of the [attribute](#_Trm00441) class T that has been initialized with the [positional\_argument\_list](#_Grm00147) P and the [named\_argument\_list](#_Grm00147) N.

## Reserved [attribute](#_Trm00441)s

A small number of [attribute](#_Trm00441)s affect the language in some way. These [attribute](#_Trm00441)s include:

* System.AttributeUsageAttribute ([§17.4.1](#_Toc00577)), which is used to describe the ways in which an [attribute](#_Trm00441) class can be used.
* System.Diagnostics.ConditionalAttribute ([§17.4.2](#_Toc00578)), which is used to define conditional [method](#_Trm00056)s.
* System.ObsoleteAttribute ([§17.4.3](#_Toc00581)), which is used to mark a member as obsolete.
* System.Runtime.CompilerServices.CallerLineNumberAttribute, System.Runtime.CompilerServices.CallerFilePathAttribute and System.Runtime.CompilerServices.CallerMemberNameAttribute ([§17.4.4](#_Toc00582)), which are used to supply information about the calling context to [optional parameter](#_Trm00334)s.

### The AttributeUsage [attribute](#_Trm00441)

The [attribute](#_Trm00441) AttributeUsage is used to describe the manner in which the [attribute](#_Trm00441) class can be used.

A class that is decorated with the AttributeUsage [attribute](#_Trm00441) 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](#_Trm00441)

The [attribute](#_Trm00441) 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 [method](#_Trm00056)s

A [method](#_Trm00056) decorated with the Conditional [attribute](#_Trm00441) is a conditional [method](#_Trm00056). The Conditional [attribute](#_Trm00441) indicates a condition by testing a conditional compilation symbol. Calls to a conditional [method](#_Trm00056) are either included or omitted depending on whether this symbol is [defined](#_Trm00121) at the point of the call. If the symbol is [defined](#_Trm00121), the call is included; otherwise, the call (including evaluation of the receiver and [parameters](#_Trm00059) of the call) is omitted.

A conditional [method](#_Trm00056) is subject to the following restrictions:

* The conditional [method](#_Trm00056) must be a [method](#_Trm00056) in a [class\_declaration](#_Grm00107) or [struct\_declaration](#_Grm00126). A compile-time error occurs if the Conditional [attribute](#_Trm00441) is specified on a [method](#_Trm00056) in an [interface](#_Trm00102) declaration.
* The conditional [method](#_Trm00056) must have a [return type](#_Trm00060) of void.
* The conditional [method](#_Trm00056) must not be marked with the override modifier. A conditional [method](#_Trm00056) may be marked with the virtual modifier, however. Overrides of such a [method](#_Trm00056) are [implicit](#_Trm00197)ly conditional, and must not be [explicit](#_Trm00198)ly marked with a Conditional [attribute](#_Trm00441).
* The conditional [method](#_Trm00056) must not be an implementation of an [interface](#_Trm00102) [method](#_Trm00056). Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional [method](#_Trm00056) is used in a [delegate\_creation\_expression](#_Grm00048). 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](#_Trm00056). Class2's Test [method](#_Trm00056) calls this [method](#_Trm00056). Since the conditional compilation symbol DEBUG is [defined](#_Trm00121), if Class2.Test is called, it will call M. If the symbol DEBUG had not been [defined](#_Trm00121), 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](#_Trm00056) 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](#_Trm00056) Class1.F, which is conditional based on whether or not DEBUG is [defined](#_Trm00121). Since this symbol is [defined](#_Trm00121) 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](#_Trm00450) in an [inheritance](#_Trm00047) chain can be confusing. Calls made to a conditional [method](#_Trm00056) through base, of the form base.M, are subject to the normal conditional [method](#_Trm00056) 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](#_Trm00121) in its base class. This call is omitted because the base [method](#_Trm00056) is conditional based on the presence of the symbol DEBUG, which is un[defined](#_Trm00121). Thus, the [method](#_Trm00056) writes to the console "Class2.M executed" only. Judicious use of [pp\_declaration](#_Grm00020)s can eliminate such problems.

#### Conditional [attribute](#_Trm00441) classes

An [attribute](#_Trm00441) class ([§17.1](#_Toc00568)) decorated with one or more Conditional [attribute](#_Trm00441)s is a ***conditional attribute class***. A conditional [attribute](#_Trm00441) class is thus associated with the conditional compilation symbols declared in its Conditional [attribute](#_Trm00441)s. This example:

using System;  
using System.Diagnostics;  
[Conditional("ALPHA")]  
[Conditional("BETA")]  
public class TestAttribute : Attribute {}

declares TestAttribute as a conditional [attribute](#_Trm00441) class associated with the conditional compilations symbols ALPHA and BETA.

[Attribute specification](#_Trm00447)s ([§17.2](#_Toc00572)) of a conditional [attribute](#_Trm00441) are included if one or more of its associated conditional compilation symbols is [defined](#_Trm00121) at the point of specification, otherwise the [attribute](#_Trm00441) specification is omitted.

It is important to note that the inclusion or exclusion of an [attribute](#_Trm00441) specification of a conditional [attribute](#_Trm00441) 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](#_Trm00441) Test, which is conditional based on whether or not DEBUG is [defined](#_Trm00121). Since this symbol is [defined](#_Trm00121) in the context of Class1 but not Class2, the specification of the Test [attribute](#_Trm00441) on Class1 is included, while the specification of the Test [attribute](#_Trm00441) on Class2 is omitted.

### The Obsolete [attribute](#_Trm00441)

The [attribute](#_Trm00441) Obsolete is used to mark [types](#_Trm00011) and [members](#_Trm00012) of [types](#_Trm00011) 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](#_Trm00109) uses a type or member that is decorated with the Obsolete [attribute](#_Trm00441), 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](#_Trm00209) false. The compiler issues an error if the error parameter is specified and has the [value](#_Trm00209) 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](#_Trm00441). 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 [attribute](#_Trm00441)s

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 [attribute](#_Trm00441)s provide a way to pass such information transparently.

When an [optional parameter](#_Trm00334) is annotated with one of the caller info [attribute](#_Trm00441)s, omitting the corresponding argument in a call does not necessarily cause the default parameter [value](#_Trm00209) to be substituted. Instead, if the specified information about the calling context is available, that information will be passed as the argument [value](#_Trm00209).

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](#_Trm00062) 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 [attribute](#_Trm00441)s can occur on [optional parameter](#_Trm00334)s anywhere, including in delegate declarations. However, the specific caller info [attribute](#_Trm00441)s have restrictions on the [types](#_Trm00011) of the [parameters](#_Trm00059) they can [attribute](#_Trm00441), so that there will always be an [implicit](#_Trm00197) [conversion](#_Trm00196) from a substituted [value](#_Trm00209) to the parameter type.

It is an error to have the same caller info [attribute](#_Trm00441) on a parameter of both the defining and implementing part of a partial [method](#_Trm00056) declaration. Only caller info [attribute](#_Trm00441)s in the defining part are applied, whereas caller info [attribute](#_Trm00441)s occurring only in the implementing part are ignored.

Caller information does not affect [overload resolution](#_Trm00078). As the [attribute](#_Trm00441)d [optional parameter](#_Trm00334)s are still omitted from the source code of the caller, [overload resolution](#_Trm00078) ignores those [parameters](#_Trm00059) in the same way it ignores other omitted [optional parameter](#_Trm00334)s ([§7.5.3](#_Toc00242)).

Caller information is only substituted when a function is [explicit](#_Trm00198)ly invoked in source code. Implicit invocations such as [implicit](#_Trm00197) 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 [attribute](#_Trm00441)d parameter is omitted in such cases, the specified [default value](#_Trm00164) 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 parameter](#_Trm00334)s with caller info [attribute](#_Trm00441)s, 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](#_Trm00441) is specified on a given parameter, they are preferred in the following order: CallerLineNumber, CallerFilePath, CallerMemberName.

#### The CallerLineNumber [attribute](#_Trm00441)

The System.Runtime.CompilerServices.CallerLineNumberAttribute is allowed on [optional parameter](#_Trm00334)s when there is a standard [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.3.1](#_Toc00191)) from the [constant](#_Trm00322) [value](#_Trm00209) int.MaxValue to the parameter's type. This ensures that any non-negative line number up to that [value](#_Trm00209) can be passed without error.

If a function invocation from a location in source code omits an [optional parameter](#_Trm00334) with the CallerLineNumberAttribute, then a numeric [literal](#_Trm00118) representing that location's line number is used as an argument to the invocation instead of the default parameter [value](#_Trm00209).

If the invocation spans multiple lines, the line chosen is implementation-dependent.

Note that the line number may be affected by #line directives ([§2.5.7](#_Toc00061)).

#### The CallerFilePath [attribute](#_Trm00441)

The System.Runtime.CompilerServices.CallerFilePathAttribute is allowed on [optional parameter](#_Trm00334)s when there is a standard [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.3.1](#_Toc00191)) from string to the parameter's type.

If a function invocation from a location in source code omits an [optional parameter](#_Trm00334) with the CallerFilePathAttribute, then a string [literal](#_Trm00118) representing that location's file path is used as an argument to the invocation instead of the default parameter [value](#_Trm00209).

The format of the file path is implementation-dependent.

Note that the file path may be affected by #line directives ([§2.5.7](#_Toc00061)).

#### The CallerMemberName [attribute](#_Trm00441)

The System.Runtime.CompilerServices.CallerMemberNameAttribute is allowed on [optional parameter](#_Trm00334)s when there is a standard [implicit](#_Trm00197) [conversion](#_Trm00196) ([§6.3.1](#_Toc00191)) 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](#_Trm00441) applied to the function member itself or its [return type](#_Trm00060), [parameters](#_Trm00059) or type [parameters](#_Trm00059) in source code omits an [optional parameter](#_Trm00334) with the CallerMemberNameAttribute, then a string [literal](#_Trm00118) representing the name of that member is used as an argument to the invocation instead of the default parameter [value](#_Trm00209).

For invocations that occur within [generic method](#_Trm00333)s, only the [method](#_Trm00056) name itself is used, without the type parameter list.

For invocations that occur within [explicit](#_Trm00198) [interface](#_Trm00102) member implementations, only the [method](#_Trm00056) name itself is used, without the preceding [interface](#_Trm00102) qualification.

For invocations that occur within [property](#_Trm00348) or [event](#_Trm00088) [accessors](#_Trm00083), the member name used is that of the [property](#_Trm00348) or [event](#_Trm00088) itself.

For invocations that occur within [indexer](#_Trm00087) [accessors](#_Trm00083), the member name used is that supplied by an IndexerNameAttribute ([§17.5.2.1](#_Toc00589)) on the [indexer](#_Trm00087) member, if present, or the default name Item otherwise.

For invocations that occur within declarations of [instance](#_Trm00172) constructors, [static constructor](#_Trm00081)s, [destructor](#_Trm00091)s and [operator](#_Trm00090)s 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 [attribute](#_Trm00441)s that enable C# [program](#_Trm00109)s to interoperate with components written using COM and Win32 DLLs. For example, the DllImport [attribute](#_Trm00441) can be used on a static extern [method](#_Trm00056) to indicate that the implementation of the [method](#_Trm00056) is to be found in a Win32 DLL. These [attribute](#_Trm00441)s are found in the System.Runtime.InteropServices namespace, and detailed documentation for these [attribute](#_Trm00441)s is found in the .NET runtime documentation.

### Interoperation with other .NET languages

#### The IndexerName [attribute](#_Trm00441)

Indexers are implemented in .NET using indexed properties, and have a name in the .NET [metadata](#_Trm00017). If no IndexerName [attribute](#_Trm00441) is present for an [indexer](#_Trm00087), then the name Item is used by default. The IndexerName [attribute](#_Trm00441) 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

The core C# language, as [defined](#_Trm00121) in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides [references](#_Trm00160) and the ability to create [object](#_Trm00173)s that are managed by a [garbage collector](#_Trm00154). 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](#_Trm00093) beyond its bounds. Whole categories of bugs that routinely plague C and C++ [program](#_Trm00109)s 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](#_Trm00011) 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](#_Trm00453) it is possible to declare and operate on pointers, to perform [conversion](#_Trm00196)s between pointers and integral [types](#_Trm00011), to take the address of [variables](#_Trm00031), and so forth. In a sense, writing [unsafe code](#_Trm00453) is much like writing C code within a C# [program](#_Trm00109).

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](#_Trm00453) 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](#_Grm00148):

* A declaration of a class, struct, [interface](#_Trm00102), 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](#_Trm00102)) is considered an unsafe context.
* A declaration of a [field](#_Trm00323), [method](#_Trm00056), [property](#_Trm00348), [event](#_Trm00088), [indexer](#_Trm00087), [operator](#_Trm00090), [instance](#_Trm00172) constructor, [destructor](#_Trm00091), or [static constructor](#_Trm00081) may include an unsafe modifier, in which case the entire textual extent of that member declaration is considered an unsafe context.
* An [unsafe\_statement](#_Grm00148) enables the use of an unsafe context within a [block](#_Grm00071). The entire textual extent of the associated [block](#_Grm00071) 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 [field](#_Trm00323)s 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](#_Trm00323) declarations cause those declarations to be considered unsafe contexts.

Other than establishing an unsafe context, thus permitting the use of pointer [types](#_Trm00011), 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](#_Trm00056) 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](#_Trm00056) in B itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the [method](#_Trm00056)'s [signature](#_Trm00061)

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](#_Trm00061) 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](#_Trm00056) declaration, as is the case in B.

## Pointer [types](#_Trm00011)

In an unsafe context, a [type](#_Grm00028) ([§4](#_Toc00090)) may be a [pointer\_type](#_Grm00150) as well as a [value\_type](#_Grm00029) or a [reference\_type](#_Grm00030). However, a [pointer\_type](#_Grm00150) may also be used in a typeof expression ([§7.6.10.6](#_Toc00276)) outside of an unsafe context as such usage is not unsafe.

type\_unsafe:  
 | pointer\_type  
 ;

A [pointer\_type](#_Grm00150) is written as an [unmanaged\_type](#_Grm00150) or the [keyword](#_Trm00117) void, followed by a \* token:

pointer\_type:  
 | unmanaged\_type '\*'  
 | 'void' '\*'  
 ;  
  
unmanaged\_type:  
 | type  
 ;

The type specified [before](#_Trm00381) 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](#_Trm00209) of the pointer type points.

Unlike [references](#_Trm00160) ([value](#_Trm00209)s of [reference types](#_Trm00019)), pointers are not tracked by the [garbage collector](#_Trm00154) -- the [garbage collector](#_Trm00154) 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](#_Trm00160), and the [referent type](#_Trm00454) of a pointer must be an [unmanaged\_type](#_Grm00150).

An [unmanaged\_type](#_Grm00150) is any type that isn't a [reference\_type](#_Grm00030) or [constructed type](#_Trm00178), and doesn't contain [reference\_type](#_Grm00030) or [constructed type](#_Trm00178) [field](#_Trm00323)s at any level of nesting. In other words, an [unmanaged\_type](#_Grm00150) is one of the following:

* sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool.
* Any [enum\_type](#_Grm00029).
* Any [pointer\_type](#_Grm00150).
* Any user-[defined](#_Trm00121) [struct\_type](#_Grm00029) that is not a [constructed type](#_Trm00178) and contains [field](#_Trm00323)s of [unmanaged\_type](#_Grm00150)s only.

The intuitive rule for mixing of pointers and [references](#_Trm00160) is that referents of [references](#_Trm00160) ([object](#_Trm00173)s) are permitted to contain pointers, but referents of pointers are not permitted to contain [references](#_Trm00160).

Some examples of pointer [types](#_Trm00011) 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](#_Trm00093) of pointers to int |
| void\* | Pointer to unknown type |

For a given implementation, all pointer [types](#_Trm00011) 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](#_Trm00106) only, not as a prefix punctuator on each pointer name. For example

int\* pi, pj; // NOT as int \*pi, \*pj;

The [value](#_Trm00209) of a pointer having type T\* represents the address of a variable of type T. The pointer indirection [operator](#_Trm00090) \* ([§18.5.1](#_Toc00597)) 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](#_Trm00173) reference, a pointer may be null. Applying the indirection [operator](#_Trm00090) to a null pointer results in implementation-[defined](#_Trm00121) behavior. A pointer with [value](#_Trm00209) null is represented by all-bits-zero.

The void\* type represents a pointer to an unknown type. Because the [referent type](#_Trm00454) is unknown, the indirection [operator](#_Trm00090) 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](#_Trm00011) are a separate category of [types](#_Trm00011). Unlike [reference types](#_Trm00019) and [value](#_Trm00209) [types](#_Trm00011), pointer [types](#_Trm00011) do not inherit from object and no [conversion](#_Trm00196)s exist between pointer [types](#_Trm00011) and object. In particular, [boxing](#_Trm00029) and un[boxing](#_Trm00029) ([§4.3](#_Toc00110)) are not supported for pointers. However, [conversion](#_Trm00196)s are permitted between different pointer [types](#_Trm00011) and between pointer [types](#_Trm00011) and the integral [types](#_Trm00011). This is described in [§18.4](#_Toc00594).

A [pointer\_type](#_Grm00150) cannot be used as a type argument ([§4.4](#_Toc00113)), and [type inference](#_Trm00231) ([§7.5.2](#_Toc00227)) fails on [generic method](#_Trm00333) calls that would have inferred a type argument to be a pointer type.

A [pointer\_type](#_Grm00150) may be used as the type of a volatile [field](#_Trm00323) ([§10.5.3](#_Toc00436)).

Although pointers can be passed as ref or out [parameters](#_Trm00059), doing so can cause un[defined](#_Trm00121) behavior, since the pointer may well be set to point to a [local variable](#_Trm00193) which no longer exists when the called [method](#_Trm00056) returns, or the fixed [object](#_Trm00173) 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](#_Trm00056) can return a [value](#_Trm00209) of some type, and that type can be a pointer. For example, when given a pointer to a contiguous [sequence](#_Trm00259) of ints, that [sequence](#_Trm00259)'s element count, and some other int [value](#_Trm00209), the following [method](#_Trm00056) returns the address of that [value](#_Trm00209) in that [sequence](#_Trm00259), 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 con[structs](#_Trm00092) are available for operating on pointers:

* The \* [operator](#_Trm00090) may be used to perform pointer indirection ([§18.5.1](#_Toc00597)).
* The -> [operator](#_Trm00090) may be used to access a member of a struct through a pointer ([§18.5.2](#_Toc00598)).
* The [] [operator](#_Trm00090) may be used to index a pointer ([§18.5.3](#_Toc00599)).
* The & [operator](#_Trm00090) may be used to obtain the address of a variable ([§18.5.4](#_Toc00600)).
* The ++ and -- [operator](#_Trm00090)s may be used to increment and decrement pointers ([§18.5.5](#_Toc00601)).
* The + and - [operator](#_Trm00090)s may be used to perform pointer arithmetic ([§18.5.6](#_Toc00602)).
* The ==, !=, <, >, <=, and => [operator](#_Trm00090)s may be used to compare pointers ([§18.5.7](#_Toc00603)).
* The stackalloc [operator](#_Trm00090) may be used to allocate memory from the call stack ([§18.7](#_Toc00606)).
* The fixed statement may be used to temporarily fix a variable so its address can be obtained ([§18.6](#_Toc00605)).

## Fixed and moveable [variables](#_Trm00031)

The address-of [operator](#_Trm00090) ([§18.5.4](#_Toc00600)) and the fixed statement ([§18.6](#_Toc00605)) divide [variables](#_Trm00031) into two categories: ***Fixed variables*** and ***moveable variables***.

[Fixed variables](#_Trm00455) reside in storage locations that are unaffected by operation of the [garbage collector](#_Trm00154). (Examples of fixed [variables](#_Trm00031) include [local variable](#_Trm00193)s, [value](#_Trm00209) [parameters](#_Trm00059), and [variables](#_Trm00031) created by dereferencing pointers.) On the other hand, [moveable variables](#_Trm00456) reside in storage locations that are subject to relocation or disposal by the [garbage collector](#_Trm00154). (Examples of [moveable variables](#_Trm00456) include [field](#_Trm00323)s in [object](#_Trm00173)s and [elements](#_Trm00094) of [array](#_Trm00093)s.)

The & [operator](#_Trm00090) ([§18.5.4](#_Toc00600)) 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](#_Trm00154), the address of a moveable variable can only be obtained using a fixed statement ([§18.6](#_Toc00605)), 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](#_Grm00036) ([§7.6.2](#_Toc00254)) that refers to a [local variable](#_Trm00193) or a [value](#_Trm00209) parameter, unless the variable is [captured](#_Trm00255) by an [anonymous function](#_Trm00253).
* A variable resulting from a [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)) of the form V.I, where V is a fixed variable of a [struct\_type](#_Grm00029).
* A variable resulting from a [pointer\_indirection\_expression](#_Grm00152) ([§18.5.1](#_Toc00597)) of the form \*P, a [pointer\_member\_access](#_Grm00153) ([§18.5.2](#_Toc00598)) of the form P->I, or a [pointer\_element\_access](#_Grm00154) ([§18.5.3](#_Toc00599)) of the form P[E].

All other [variables](#_Trm00031) are classified as [moveable variables](#_Trm00456).

Note that a static [field](#_Trm00323) 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 [conversion](#_Trm00196)s

In an unsafe context, the set of available [implicit](#_Trm00197) [conversion](#_Trm00196)s ([§6.1](#_Toc00168)) is extended to include the following [implicit](#_Trm00197) pointer [conversion](#_Trm00196)s:

* From any [pointer\_type](#_Grm00150) to the type void\*.
* From the null [literal](#_Trm00118) to any [pointer\_type](#_Grm00150).

Additionally, in an unsafe context, the set of available [explicit](#_Trm00198) [conversion](#_Trm00196)s ([§6.2](#_Toc00181)) is extended to include the following [explicit](#_Trm00198) pointer [conversion](#_Trm00196)s:

* From any [pointer\_type](#_Grm00150) to any other [pointer\_type](#_Grm00150).
* From sbyte, byte, short, ushort, int, uint, long, or ulong to any [pointer\_type](#_Grm00150).
* From any [pointer\_type](#_Grm00150) to sbyte, byte, short, ushort, int, uint, long, or ulong.

Finally, in an unsafe context, the set of standard [implicit](#_Trm00197) [conversion](#_Trm00196)s ([§6.3.1](#_Toc00191)) includes the following pointer [conversion](#_Trm00196):

* From any [pointer\_type](#_Grm00150) to the type void\*.

Conversions between two pointer [types](#_Trm00011) never change the actual pointer [value](#_Trm00209). In other words, a [conversion](#_Trm00196) 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 un[defined](#_Trm00121) 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](#_Trm00056) displays each of the eight bytes in a double as a hexadecimal [value](#_Trm00209):

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](#_Trm00306) endianness.

Mappings between pointers and integers are implementation-[defined](#_Trm00121). However, on 32\* and 64-bit CPU architectures with a linear address space, [conversion](#_Trm00196)s of pointers to or from integral [types](#_Trm00011) typically behave exactly like [conversion](#_Trm00196)s of uint or ulong [value](#_Trm00209)s, respectively, to or from those integral [types](#_Trm00011).

### Pointer [array](#_Trm00093)s

In an unsafe context, [array](#_Trm00093)s of pointers can be constructed. Only some of the [conversion](#_Trm00196)s that apply to other [array](#_Trm00093) [types](#_Trm00011) are allowed on pointer [array](#_Trm00093)s:

* The [implicit](#_Trm00197) reference [conversion](#_Trm00196) ([§6.1.6](#_Toc00174)) from any [array\_type](#_Grm00030) to System.Array and the [interface](#_Trm00102)s it implements also applies to pointer [array](#_Trm00093)s. However, any attempt to access the [array](#_Trm00093) [elements](#_Trm00094) through System.Array or the [interface](#_Trm00102)s it implements will result in an exception at run-time, as pointer [types](#_Trm00011) are not convertible to object.
* The [implicit](#_Trm00197) and [explicit](#_Trm00198) reference [conversion](#_Trm00196)s ([§6.1.6](#_Toc00174), [§6.2.4](#_Toc00185)) from a single-dimensional [array](#_Trm00093) type S[] to System.Collections.Generic.IList<T> and its generic [base interfaces](#_Trm00430) never apply to pointer [array](#_Trm00093)s, since pointer [types](#_Trm00011) cannot be used as type [arguments](#_Trm00062), and there are no [conversion](#_Trm00196)s from pointer [types](#_Trm00011) to non-pointer [types](#_Trm00011).
* The [explicit](#_Trm00198) reference [conversion](#_Trm00196) ([§6.2.4](#_Toc00185)) from System.Array and the [interface](#_Trm00102)s it implements to any [array\_type](#_Grm00030) applies to pointer [array](#_Trm00093)s.
* The [explicit](#_Trm00198) reference [conversion](#_Trm00196)s ([§6.2.4](#_Toc00185)) from System.Collections.Generic.IList<S> and its [base interfaces](#_Trm00430) to a single-dimensional [array](#_Trm00093) type T[] never applies to pointer [array](#_Trm00093)s, since pointer [types](#_Trm00011) cannot be used as type [arguments](#_Trm00062), and there are no [conversion](#_Trm00196)s from pointer [types](#_Trm00011) to non-pointer [types](#_Trm00011).

These restrictions mean that the expansion for the foreach statement over [array](#_Trm00093)s described in [§8.8.4](#_Toc00365) cannot be applied to pointer [array](#_Trm00093)s. Instead, a foreach statement of the form

foreach (V v in x) embedded\_statement

where the type of x is an [array](#_Trm00093) 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](#_Trm00143) for-loops as follows:

{  
 T[,,...,] a = x;  
 V v;  
 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)a.GetValue(i0,i1,...,iN);  
 embedded\_statement  
 }  
}

The [variables](#_Trm00031) a, i0, i1, ..., iN are not [visible](#_Trm00152) to or [accessible](#_Trm00138) to x or the [embedded\_statement](#_Grm00070) or any other source code of the [program](#_Trm00109). The variable v is [read-only](#_Trm00355) in the embedded statement. If there is not an [explicit](#_Trm00198) [conversion](#_Trm00196) ([§18.4](#_Toc00594)) from T (the [element type](#_Trm00095)) to V, an error is produced and no further steps are taken. If x has the [value](#_Trm00209) 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](#_Grm00036) ([§7.6.2](#_Toc00254)), [member\_access](#_Grm00038) ([§7.6.4](#_Toc00257)), [invocation\_expression](#_Grm00039) ([§7.6.5](#_Toc00260)), or [element\_access](#_Grm00040) ([§7.6.6](#_Toc00264)) is of a pointer type.

In an unsafe context, the [primary\_no\_array\_creation\_expression](#_Grm00035) ([§7.6](#_Toc00252)) and [unary\_expression](#_Grm00053) ([§7.7](#_Toc00281)) productions permit the following additional con[structs](#_Trm00092):

primary\_no\_array\_creation\_expression\_unsafe:  
 | pointer\_member\_access  
 | pointer\_element\_access  
 | sizeof\_expression  
 ;  
  
unary\_expression\_unsafe:  
 | pointer\_indirection\_expression  
 | addressof\_expression  
 ;

These con[structs](#_Trm00092) are described in the following sections. The [precedence](#_Trm00035) and [associativity](#_Trm00218) of the unsafe [operator](#_Trm00090)s is implied by the grammar.

### Pointer indirection

A [pointer\_indirection\_expression](#_Grm00152) consists of an asterisk (\*) followed by a [unary\_expression](#_Grm00053).

pointer\_indirection\_expression:  
 | '\*' unary\_expression  
 ;

The unary \* [operator](#_Trm00090) 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](#_Trm00090) to an expression of type void\* or to an expression that isn't of a pointer type.

The effect of applying the unary \* [operator](#_Trm00090) to a null pointer is implementation-[defined](#_Trm00121). In particular, there is no guarantee that this operation throws a System.NullReferenceException.

If an invalid [value](#_Trm00209) has been assigned to the pointer, the behavior of the unary \* [operator](#_Trm00090) is un[defined](#_Trm00121). Among the invalid [value](#_Trm00209)s for dereferencing a pointer by the unary \* [operator](#_Trm00090) are an address inappropriately aligned for the type pointed to (see example in [§18.4](#_Toc00594)), and the address of a variable [after](#_Trm00384) 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](#_Trm00186) ([§5.3.1](#_Toc00133)).

### Pointer member access

A [pointer\_member\_access](#_Grm00153) consists of a [primary\_expression](#_Grm00035), followed by a "->" token, followed by an [identifier](#_Grm00007) and an optional [type\_argument\_list](#_Grm00031).

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](#_Trm00138) 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](#_Trm00090) (\*), see [§18.5.1](#_Toc00597). For a description of the member access [operator](#_Trm00090) (.), see [§7.6.4](#_Toc00257).

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](#_Trm00090) is used to access [field](#_Trm00323)s and invoke a [method](#_Trm00056) of a struct through a pointer. Because the operation P->I is precisely equivalent to (\*P).I, the Main [method](#_Trm00056) 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](#_Grm00154) consists of a [primary\_no\_array\_creation\_expression](#_Grm00035) 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 [implicit](#_Trm00197)ly 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](#_Trm00090) (\*), see [§18.5.1](#_Toc00597). For a description of the pointer addition [operator](#_Trm00090) (+), see [§18.5.6](#_Toc00602).

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](#_Trm00090) does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is un[defined](#_Trm00121). This is the same as C and C++.

### The address-of [operator](#_Trm00090)

An [addressof\_expression](#_Grm00155) consists of an ampersand (&) followed by a [unary\_expression](#_Grm00053).

addressof\_expression:  
 | '&' unary\_expression  
 ;

Given an expression E which is of a type T and is classified as a fixed variable ([§18.3](#_Toc00593)), 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](#_Trm00209). A compile-time error occurs if E is not classified as a variable, if E is classified as a [read-only](#_Trm00355) [local variable](#_Trm00193), or if E denotes a moveable variable. In the last case, a fixed statement ([§18.6](#_Toc00605)) can be used to temporarily "fix" the variable [before](#_Trm00381) obtaining its address. As stated in [§7.6.4](#_Toc00257), outside an [instance](#_Trm00172) constructor or [static constructor](#_Trm00081) for a struct or class that defines a readonly [field](#_Trm00323), that [field](#_Trm00323) is considered a [value](#_Trm00209), not a variable. As such, its address cannot be taken. Similarly, the address of a [constant](#_Trm00322) cannot be taken.

The & [operator](#_Trm00090) does not require its argument to be [definitely assigned](#_Trm00068), but following an & operation, the variable to which the [operator](#_Trm00090) is applied is considered [definitely assigned](#_Trm00068) in the execution path in which the operation occurs. It is the responsibility of the [program](#_Trm00109)mer 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](#_Trm00068) 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 [program](#_Trm00109)mer, and no compile-time error would occur if the assignment was removed.

The rules of definite assignment for the & [operator](#_Trm00090) exist such that redundant initialization of [local variable](#_Trm00193)s 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 -- [operator](#_Trm00090)s ([§7.6.9](#_Toc00269) and [§7.7.5](#_Toc00286)) can be applied to pointer [variables](#_Trm00031) of all [types](#_Trm00011) except void\*. Thus, for every pointer type T\*, the following [operator](#_Trm00090)s are [implicit](#_Trm00197)ly [defined](#_Trm00121):

T\* operator ++(T\* x);  
T\* operator --(T\* x);

The [operator](#_Trm00090)s produce the same results as x + 1 and x - 1, respectively ([§18.5.6](#_Toc00602)). In other words, for a pointer variable of type T\*, the ++ [operator](#_Trm00090) adds sizeof(T) to the address contained in the variable, and the -- [operator](#_Trm00090) 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](#_Trm00121), but no exceptions are produced.

### Pointer arithmetic

In an unsafe context, the + and - [operator](#_Trm00090)s ([§7.8.4](#_Toc00296) and [§7.8.5](#_Toc00297)) can be applied to [value](#_Trm00209)s of all pointer [types](#_Trm00011) except void\*. Thus, for every pointer type T\*, the following [operator](#_Trm00090)s are [implicit](#_Trm00197)ly [defined](#_Trm00121):

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](#_Trm00209) 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](#_Trm00209) 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](#_Trm00121) fashion, but no exceptions are produced.

### Pointer comparison

In an unsafe context, the ==, !=, <, >, <=, and => [operator](#_Trm00090)s ([§7.10](#_Toc00299)) can be applied to [value](#_Trm00209)s of all pointer [types](#_Trm00011). The pointer [comparison operators](#_Trm00252) 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](#_Trm00197) [conversion](#_Trm00196) exists from any pointer type to the void\* type, [operands](#_Trm00033) of any pointer type can be compared using these [operator](#_Trm00090)s. The [comparison operators](#_Trm00252) compare the addresses given by the two [operands](#_Trm00033) as if they were unsigned integers.

### The sizeof [operator](#_Trm00090)

The sizeof [operator](#_Trm00090) 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](#_Grm00150) ([§18.2](#_Toc00592)).

sizeof\_expression:  
 | 'sizeof' '(' unmanaged\_type ')'  
 ;

The result of the sizeof [operator](#_Trm00090) is a [value](#_Trm00209) of type int. For certain pre[defined](#_Trm00121) [types](#_Trm00011), the sizeof [operator](#_Trm00090) yields a [constant](#_Trm00322) [value](#_Trm00209) 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](#_Trm00011), the result of the sizeof [operator](#_Trm00090) is implementation-[defined](#_Trm00121) and is classified as a [value](#_Trm00209), not a [constant](#_Trm00322).

The order in which [members](#_Trm00012) 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](#_Grm00070) ([§8](#_Toc00348)) production permits an additional construct, the fixed statement, which is used to "fix" a moveable variable such that its address remains [constant](#_Trm00322) 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](#_Grm00157) declares a [local variable](#_Trm00193) of the given [pointer\_type](#_Grm00150) and initializes that [local variable](#_Trm00193) with the address computed by the corresponding [fixed\_pointer\_initializer](#_Grm00157). A [local variable](#_Trm00193) declared in a fixed statement is [accessible](#_Trm00138) in any [fixed\_pointer\_initializer](#_Grm00157)s occurring to the right of that variable's declaration, and in the [embedded\_statement](#_Grm00070) of the fixed statement. A [local variable](#_Trm00193) declared by a fixed statement is considered [read-only](#_Trm00355). A compile-time error occurs if the embedded statement attempts to modify this [local variable](#_Trm00193) (via assignment or the ++ and -- [operator](#_Trm00090)s) or pass it as a ref or out parameter.

A [fixed\_pointer\_initializer](#_Grm00157) can be one of the following:

* The token "&" followed by a [variable\_reference](#_Grm00033) ([§5.3.3](#_Toc00135)) to a moveable variable ([§18.3](#_Toc00593)) of an unmanaged type T, provided the type T\* is [implicit](#_Trm00197)ly 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](#_Grm00030) with [elements](#_Trm00094) of an unmanaged type T, provided the type T\* is [implicit](#_Trm00197)ly 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](#_Trm00093), and the entire [array](#_Trm00093) is guaranteed to remain at a fixed address for the duration of the fixed statement. The behavior of the fixed statement is implementation-[defined](#_Trm00121) if the [array](#_Trm00093) expression is null or if the [array](#_Trm00093) has zero [elements](#_Trm00094).
* An expression of type string, provided the type char\* is [implicit](#_Trm00197)ly 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](#_Trm00121) if the string expression is null.
* A [simple\_name](#_Grm00036) or [member\_access](#_Grm00038) that [references](#_Trm00160) a fixed size buffer member of a moveable variable, provided the type of the fixed size buffer member is [implicit](#_Trm00197)ly 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 ([§18.7.2](#_Toc00608)), 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](#_Grm00157) the fixed statement ensures that the variable referenced by the address is not subject to relocation or disposal by the [garbage collector](#_Trm00154) for the duration of the fixed statement. For example, if the address computed by a [fixed\_pointer\_initializer](#_Grm00157) [references](#_Trm00160) a [field](#_Trm00323) of an [object](#_Trm00173) or an element of an [array](#_Trm00093) [instance](#_Trm00172), the fixed statement guarantees that the containing [object](#_Trm00173) [instance](#_Trm00172) is not relocated or disposed of during the lifetime of the statement.

It is the [program](#_Trm00109)mer's responsibility to ensure that pointers created by fixed [statements](#_Trm00037) do not survive beyond execution of those [statements](#_Trm00037). For example, when pointers created by fixed [statements](#_Trm00037) are passed to external APIs, it is the [program](#_Trm00109)mer's responsibility to ensure that the APIs retain no memory of these pointers.

Fixed [object](#_Trm00173)s may cause fragmentation of the heap (because they can't be moved). For that reason, [object](#_Trm00173)s 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](#_Trm00323), the second statement fixes and obtains the address of an [instance](#_Trm00172) [field](#_Trm00323), and the third statement fixes and obtains the address of an [array](#_Trm00093) element. In each case it would have been an error to use the regular & [operator](#_Trm00090) since the [variables](#_Trm00031) are all classified as [moveable variables](#_Trm00456).

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](#_Trm00093) [elements](#_Trm00094) of single-dimensional [array](#_Trm00093)s are stored in increasing index order, starting with index 0 and ending with index Length - 1. For multi-dimensional [array](#_Trm00093)s, [array](#_Trm00093) [elements](#_Trm00094) 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](#_Trm00093) [instance](#_Trm00172) a, the pointer [value](#_Trm00209)s ranging from p to p + a.Length - 1 represent addresses of the [elements](#_Trm00094) in the [array](#_Trm00093). Likewise, the [variables](#_Trm00031) ranging from p[0] to p[a.Length - 1] represent the actual [array](#_Trm00093) [elements](#_Trm00094). Given the way in which [array](#_Trm00093)s are stored, we can treat an [array](#_Trm00093) 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](#_Trm00093) so its address can be passed to a [method](#_Trm00056) 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](#_Trm00209) produced by fixing a string [instance](#_Trm00172) always points to a null-terminated string. Within a fixed statement that obtains a pointer p to a string [instance](#_Trm00172) s, the pointer [value](#_Trm00209)s ranging from p to p + s.Length - 1 represent addresses of the characters in the string, and the pointer [value](#_Trm00209) p + s.Length always points to a null character (the character with [value](#_Trm00209) '\0').

Modifying [object](#_Trm00173)s of managed type through fixed pointers can results in un[defined](#_Trm00121) behavior. For example, because strings are immutable, it is the [program](#_Trm00109)mer'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](#_Trm00172) 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 [array](#_Trm00093)s as [members](#_Trm00012) of [structs](#_Trm00092), 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](#_Trm00096) buffer of [variables](#_Trm00031) of a given type. A [fixed size buffer](#_Trm00457) declaration introduces one or more [fixed size buffer](#_Trm00457)s of a given [element type](#_Trm00095). Fixed size buffers are only permitted in struct declarations and can only occur in unsafe contexts ([§18.1](#_Toc00591)).

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](#_Trm00457) declaration may include a set of [attribute](#_Trm00441)s ([§17](#_Toc00567)), a new modifier ([§10.2.2](#_Toc00403)), a valid combination of the four access modifiers ([§10.2.3](#_Toc00404)) and an unsafe modifier ([§18.1](#_Toc00591)). The [attribute](#_Trm00441)s and modifiers apply to all of the [members](#_Trm00012) declared by the [fixed size buffer](#_Trm00457) declaration. It is an error for the same modifier to appear multiple times in a [fixed size buffer](#_Trm00457) declaration.

A [fixed size buffer](#_Trm00457) declaration is not permitted to include the static modifier.

The buffer [element type](#_Trm00095) of a [fixed size buffer](#_Trm00457) declaration specifies the [element type](#_Trm00095) of the buffer(s) introduced by the declaration. The buffer [element type](#_Trm00095) must be one of the pre[defined](#_Trm00121) [types](#_Trm00011) sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or bool.

The buffer [element type](#_Trm00095) is followed by a list of [fixed size buffer](#_Trm00457) declarators, each of which introduces a new member. A [fixed size buffer](#_Trm00457) declarator consists of an identifier that names the member, followed by a [constant](#_Trm00322) expression enclosed in [ and ] tokens. The [constant](#_Trm00322) expression denotes the number of [elements](#_Trm00094) in the member introduced by that [fixed size buffer](#_Trm00457) declarator. The type of the [constant](#_Trm00322) expression must be [implicit](#_Trm00197)ly convertible to type int, and the [value](#_Trm00209) must be a non-zero positive integer.

The [elements](#_Trm00094) of a [fixed size buffer](#_Trm00457) are guaranteed to be laid out sequentially in memory.

A [fixed size buffer](#_Trm00457) declaration that declares multiple [fixed size buffer](#_Trm00457)s is equivalent to multiple declarations of a single [fixed size buffer](#_Trm00457) declation with the same [attribute](#_Trm00441)s, and [element type](#_Trm00095)s. 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 ([§7.3](#_Toc00211)) of a [fixed size buffer](#_Trm00457) member proceeds exactly like member lookup of a [field](#_Trm00323).

A [fixed size buffer](#_Trm00457) can be referenced in an expression using a [simple\_name](#_Grm00036) ([§7.5.2](#_Toc00227)) or a [member\_access](#_Grm00038) ([§7.5.4](#_Toc00249)).

When a [fixed size buffer](#_Trm00457) 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](#_Trm00457) 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](#_Trm00209), a compile-time error occurs.
* Otherwise, if E is a moveable variable ([§18.3](#_Toc00593)) and the expression E.I is not a [fixed\_pointer\_initializer](#_Grm00157) ([§18.6](#_Toc00605)), a compile-time error occurs.
* Otherwise, E [references](#_Trm00160) a fixed variable and the result of the expression is a pointer to the first element of the [fixed size buffer](#_Trm00457) member I in E. The result is of type S\*, where S is the [element type](#_Trm00095) of I, and is classified as a [value](#_Trm00209).

The subsequent [elements](#_Trm00094) of the [fixed size buffer](#_Trm00457) can be accessed using pointer operations from the first element. Unlike access to [array](#_Trm00093)s, access to the [elements](#_Trm00094) of a [fixed size buffer](#_Trm00457) is an unsafe operation and is not range checked.

The following example declares and uses a struct with a [fixed size buffer](#_Trm00457) 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 ([§5.3](#_Toc00132)), and [fixed size buffer](#_Trm00457) [members](#_Trm00012) are ignored for purposes of definite assignment checking of struct type [variables](#_Trm00031).

When the outermost containing struct variable of a [fixed size buffer](#_Trm00457) member is a [static variable](#_Trm00188), an [instance](#_Trm00172) variable of a class [instance](#_Trm00172), or an [array](#_Trm00093) element, the [elements](#_Trm00094) of the [fixed size buffer](#_Trm00457) are automatically initialized to their [default value](#_Trm00164)s ([§5.2](#_Toc00131)). In all other cases, the initial content of a [fixed size buffer](#_Trm00457) is un[defined](#_Trm00121).

## Stack allocation

In an unsafe context, a [local variable](#_Trm00193) declaration ([§8.5.1](#_Toc00355)) 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](#_Grm00150) indicates the type of the items that will be stored in the newly allocated location, and the [expression](#_Grm00067) 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](#_Grm00068) that evaluates to a negative [value](#_Trm00209).

A stack allocation initializer of the form stackalloc T[E] requires T to be an unmanaged type ([§18.2](#_Toc00592)) 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](#_Trm00038). If E is a negative [value](#_Trm00209), then the behavior is un[defined](#_Trm00121). If E is zero, then no allocation is made, and the pointer returned is implementation-[defined](#_Trm00121). If there is not enough memory available to allocate a [block](#_Trm00038) of the given size, a System.StackOverflowException is thrown.

The content of the newly allocated memory is un[defined](#_Trm00121).

Stack allocation initializers are not permitted in catch or finally [block](#_Trm00038)s ([§8.10](#_Toc00372)).

There is no way to [explicit](#_Trm00198)ly free memory allocated using stackalloc. All stack allocated memory [block](#_Trm00038)s 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](#_Trm00056) to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the [method](#_Trm00056) returns.

## Dynamic memory allocation

Except for the stackalloc [operator](#_Trm00090), C# provides no pre[defined](#_Trm00121) con[structs](#_Trm00092) for managing non-garbage collected memory. Such services are typically provided by supporting class [libraries](#_Trm00015) 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;  
 }  
 }  
  
 // 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](#_Trm00038) with [value](#_Trm00209)s increasing from 0 to 255. It then allocates a 256 element byte [array](#_Trm00093) and uses Memory.Copy to copy the contents of the memory [block](#_Trm00038) into the byte [array](#_Trm00093). Finally, the memory [block](#_Trm00038) is freed using Memory.Free and the contents of the byte [array](#_Trm00093) are output on the console.

# Documentation comments

C# provides a mechanism for [program](#_Trm00109)mers 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](#_Trm00094), which they precede. Comments using such syntax are called ***documentation comments***. They must immediately precede a user-[defined](#_Trm00121) type (such as a class, delegate, or [interface](#_Trm00102)) or a member (such as a [field](#_Trm00323), [event](#_Trm00088), [property](#_Trm00348), or [method](#_Trm00056)). 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](#_Trm00459) is called the ***documentation file***. A [documentation file](#_Trm00460) 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](#_Trm00458), 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](#_Trm00094), 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](#_Trm00121) type (such as a class, delegate, or [interface](#_Trm00102)) or a member (such as a [field](#_Trm00323), [event](#_Trm00088), [property](#_Trm00348), or [method](#_Trm00056)) that they annotate. Attribute sections ([§17.2](#_Toc00572)) are considered part of declarations, so [documentation comments](#_Trm00458) must precede [attribute](#_Trm00441)s 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](#_Grm00160), if there is a [whitespace](#_Grm00004) character following the /// characters on each of the [single\_line\_doc\_comment](#_Grm00160)s adjacent to the current [single\_line\_doc\_comment](#_Grm00160), then that [whitespace](#_Grm00004) 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](#_Trm00384), as well as [before](#_Trm00381), 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](#_Trm00458) 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](#_Trm00460) 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](#_Trm00121) in [§19.2](#_Toc00614). Some of the recommended tags have special meanings:

* The <param> tag is used to describe [parameters](#_Trm00059). If such a tag is used, the [documentation generator](#_Trm00459) must verify that the specified parameter exists and that all [parameters](#_Trm00059) are described in [documentation comments](#_Trm00458). If such verification fails, the [documentation generator](#_Trm00459) issues a warning.
* The cref [attribute](#_Trm00441) can be attached to any tag to provide a reference to a code element. The [documentation generator](#_Trm00459) must verify that this code element exists. If the verification fails, the [documentation generator](#_Trm00459) issues a warning. When looking for a name described in a cref [attribute](#_Trm00441), the [documentation generator](#_Trm00459) must respect namespace visibility according to using [statements](#_Trm00037) appearing within the source code. For code [elements](#_Trm00094) 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&lt;T&gt;").
* The <summary> tag is intended to be used by a [documentation viewer](#_Trm00461) 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](#_Trm00460) does not provide full information about the type and [members](#_Trm00012) (for example, it does not contain any type information). To get such information about a type or member, the [documentation file](#_Trm00460) must be used in conjunction with reflection on the actual type or member.

## Recommended tags

The [documentation generator](#_Trm00459) 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> | [§19.2.1](#_Toc00615) | Set text in a code-like font |
| <code> | [§19.2.2](#_Toc00616) | Set one or more lines of source code or [program](#_Trm00109) output |
| <example> | [§19.2.3](#_Toc00617) | Indicate an example |
| <exception> | [§19.2.4](#_Toc00618) | Identifies the exceptions a [method](#_Trm00056) can throw |
| <include> | [§19.2.5](#_Toc00619) | Includes XML from an external file |
| <list> | [§19.2.6](#_Toc00620) | Create a list or table |
| <para> | [§19.2.7](#_Toc00621) | Permit structure to be added to text |
| <param> | [§19.2.8](#_Toc00622) | Describe a parameter for a [method](#_Trm00056) or constructor |
| <paramref> | [§19.2.9](#_Toc00623) | Identify that a word is a parameter name |
| <permission> | [§19.2.10](#_Toc00624) | Document the security accessibility of a member |
| <remark> | [§19.2.11](#_Toc00625) | Describe additional information about a type |
| <returns> | [§19.2.12](#_Toc00626) | Describe the return [value](#_Trm00209) of a [method](#_Trm00056) |
| <see> | [§19.2.13](#_Toc00627) | Specify a link |
| <seealso> | [§19.2.14](#_Toc00628) | Generate a See Also entry |
| <summary> | [§19.2.15](#_Toc00629) | Describe a type or a member of a type |
| <value> | [§19.2.16](#_Toc00630) | Describe a [property](#_Trm00348) |
| <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](#_Trm00038) of code. For lines of actual code, use <code> ([§19.2.2](#_Toc00616)).

**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](#_Trm00109) output in some special font. For small code fragments in narrative, use <c> ([§19.2.1](#_Toc00615)).

**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](#_Trm00056) or other library member may be used. Ordinarily, this would also involve use of the tag <code> ([§19.2.2](#_Toc00616)) as well.

**Syntax:**

<example>description</example>

**Example:**

See <code> ([§19.2.2](#_Toc00616)) for an example.

### <exception>

This tag provides a way to document the exceptions a [method](#_Trm00056) can throw.

**Syntax:**

<exception cref="member">description</exception>

where

* member is the name of a member. The [documentation generator](#_Trm00459) checks that the given member exists and translates member to the canonical element name in the [documentation file](#_Trm00460).
* 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](#_Trm00038) 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](#_Trm00038). 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> ([§19.2.11](#_Toc00625)) or <returns> ([§19.2.12](#_Toc00626)), 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-trvial 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](#_Trm00056), constructor, or [indexer](#_Trm00087).

**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](#_Trm00460) 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](#_Trm00459) checks that the given code element exists and translates *member* to the canonical element name in the [documentation file](#_Trm00460).
* 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> ([§19.2.15](#_Toc00629)) to describe the type itself and the [members](#_Trm00012) 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](#_Trm00209) of a [method](#_Trm00056).

**Syntax:**

<returns>description</returns>

where description is a description of the return [value](#_Trm00209).

**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> ([§19.2.14](#_Toc00628)) 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](#_Trm00459) checks that the given code element exists and changes *member* to the element name in the generated [documentation file](#_Trm00460).

**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> ([§19.2.13](#_Toc00627)) to specify a link from within text.

**Syntax:**

<seealso cref="member"/>

where member is the name of a member. The [documentation generator](#_Trm00459) checks that the given code element exists and changes *member* to the element name in the generated [documentation file](#_Trm00460).

**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> ([§19.2.11](#_Toc00625)) 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](#_Trm00348) to be described.

**Syntax:**

<value>property description</value>

where property description is a description for the [property](#_Trm00348).

**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](#_Trm00102), delegate, or [method](#_Trm00056).

**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](#_Trm00460) 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](#_Trm00460)

The [documentation generator](#_Trm00459) 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](#_Trm00461) can use an ID string to identify the corresponding [metadata](#_Trm00017)/reflection item to which the documentation applies.

The [documentation file](#_Trm00460) 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](#_Trm00459) 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](#_Trm00012) are [defined](#_Trm00121):

|  |  |
| --- | --- |
| **Character** | **Description** |
| E | Event |
| F | Field |
| M | Method (including constructors, [destructor](#_Trm00091)s, and [operator](#_Trm00090)s) |
| N | Namespace |
| P | Property (including [indexer](#_Trm00087)s) |
| T | Type (such as class, delegate, enum, [interface](#_Trm00102), and struct) |
| ! | Error string; the rest of the string provides information about the error. For example, the [documentation generator](#_Trm00459) generates error information for links that cannot be resolved. |

* The second part of the string is the [fully qualified name](#_Trm00153) 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 [method](#_Trm00056)s and properties with [arguments](#_Trm00062), the argument list follows, enclosed in parentheses. For those without [arguments](#_Trm00062), the parentheses are omitted. The [arguments](#_Trm00062) are separated by commas. The encoding of each argument is the same as a CLI [signature](#_Trm00061), as follows:
  + Arguments are represented by their documentation name, which is based on their [fully qualified name](#_Trm00153), modified as follows:
    - Arguments that represent [generic types](#_Trm00158) have an appended "'" character followed by the number of type [parameters](#_Trm00059)
    - Arguments having the out or ref modifier have an @ following their type name. Arguments passed by [value](#_Trm00209) or via params have no special notation.
    - Arguments that are [array](#_Trm00093)s are represented as [lowerbound:size, ... , lowerbound:size] where the number of commas is the [rank](#_Trm00099) 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 [array](#_Trm00093)s are represented by one "[]" per level.
    - Arguments that have pointer [types](#_Trm00011) 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](#_Trm00059) [defined](#_Trm00121) on [types](#_Trm00011) are encoded using the "`" character followed by the zero-based index of the type parameter.
    - Arguments that use generic type [parameters](#_Trm00059) [defined](#_Trm00121) in [method](#_Trm00056)s use a double-backtick "``" instead of the "`" used for [types](#_Trm00011).
    - Arguments that refer to constructed [generic types](#_Trm00158) are encoded using the generic type, followed by "{", followed by a comma-separated list of type [arguments](#_Trm00062), 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](#_Trm00153), 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](#_Trm00153):

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](#_Trm00082) and [indexer](#_Trm00087)s.

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 [operator](#_Trm00090)s.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x) {...}  
 }  
}  
  
"M:Acme.Widget.op\_UnaryPlus(Acme.Widget)"

The complete set of unary [operator](#_Trm00090) function names used is as follows: op\_UnaryPlus, op\_UnaryNegation, op\_LogicalNot, op\_OnesComplement, op\_Increment, op\_Decrement, op\_True, and op\_False.

* Binary [operator](#_Trm00090)s.

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](#_Trm00090) 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 [operator](#_Trm00090)s have a trailing "~" followed by the [return type](#_Trm00060).

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-trvial 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](#_Trm00459) 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-trvial maintenance has  
 been performed on the Point class.</para></summary>  
 </member>  
  
 <member name="P:Graphics.Point.X">  
 <value>Property <c>X</c> represents the point's  
 x-coordinate.</value>  
 </member>  
  
 <member name="P:Graphics.Point.Y">  
 <value>Property <c>Y</c> represents the point's  
 y-coordinate.</value>  
 </member>  
 </members>  
</doc>