

Contents

[C# documentation](#)

[Get started](#)

[Introduction](#)

[Types](#)

[Program building blocks](#)

[Major language areas](#)

[Tutorials](#)

[Choose your first lesson](#)

[Browser based tutorials](#)

[Hello world](#)

[Numbers in C#](#)

[Branches and loops](#)

[List collections](#)

[Work in your local environment](#)

[Set up your environment](#)

[Numbers in C#](#)

[Branches and loops](#)

[List collections](#)

[What's new in C#](#)

[C# 9.0](#)

[C# 8.0](#)

[C# 7.0-7.3](#)

[Compiler breaking changes](#)

[C# Version History](#)

[Relationships to .NET library](#)

[Version compatibility](#)

[Tutorials](#)

[Explore record types](#)

[Explore top level statements](#)

- Explore patterns in objects
- Safely update interfaces with default interface methods
- Create mixin functionality with default interface methods
- Explore indexes and ranges
- Work with nullable reference types
- Upgrade an app to nullable reference types
- Generate and consume asynchronous streams

Tutorials

- Introduction to classes
- Object-Oriented programming
- Explore string interpolation - interactive
- Explore string interpolation - in your environment
- Advanced scenarios for string Interpolation
- Console Application
- REST Client
- Inheritance in C# and .NET
- Work with LINQ
- Use Attributes
- Build data-driven algorithms with pattern matching
- C# concepts
 - C# type system
 - Nullable reference types
 - Choose a strategy for enabling nullable reference types
- Namespaces
- Basic Types
- Classes
- Deconstructing tuples and other types
- Interfaces
- Methods
- Properties
- Indexers
- Discards

[Generics](#)

[Iterators](#)

[Delegates & events](#)

[Introduction to Delegates](#)

[System.Delegate and the delegate keyword](#)

[Strongly Typed Delegates](#)

[Common Patterns for Delegates](#)

[Introduction to events](#)

[Standard .NET event patterns](#)

[The Updated .NET Event Pattern](#)

[Distinguishing Delegates and Events](#)

[Language-Integrated Query \(LINQ\)](#)

[Overview of LINQ](#)

[Query expression basics](#)

[LINQ in C#](#)

[Write LINQ queries in C#](#)

[Query a collection of objects](#)

[Return a query from a method](#)

[Store the results of a query in memory](#)

[Group query results](#)

[Create a nested group](#)

[Perform a subquery on a grouping operation](#)

[Group results by contiguous keys](#)

[Dynamically specify predicate filters at runtime](#)

[Perform inner joins](#)

[Perform grouped joins](#)

[Perform left outer joins](#)

[Order the results of a join clause](#)

[Join by using composite keys](#)

[Perform custom join operations](#)

[Handle null values in query expressions](#)

[Handle exceptions in query expressions](#)

[Pattern Matching](#)

[Write safe, efficient code](#)

[Expression trees](#)

[Introduction to expression trees](#)

[Expression Trees Explained](#)

[Framework Types Supporting Expression Trees](#)

[Executing Expressions](#)

[Interpreting Expressions](#)

[Building Expressions](#)

[Translating Expressions](#)

[Summary](#)

[Native interoperability](#)

[Documenting your code](#)

[Versioning](#)

[How-to C# articles](#)

[Article index](#)

[Split strings into substrings](#)

[Concatenate strings](#)

[Search strings](#)

[Modify string contents](#)

[Compare strings](#)

[Safely cast using pattern matching and is/as operators](#)

[The .NET Compiler Platform SDK \(Roslyn APIs\)](#)

[The .NET Compiler Platform SDK \(Roslyn APIs\) overview](#)

[Understand the compiler API model](#)

[Work with syntax](#)

[Work with semantics](#)

[Work with a workspace](#)

[Explore code with the syntax visualizer](#)

[Quick starts](#)

[Syntax analysis](#)

[Semantic analysis](#)

Syntax Transformation

Tutorials

Build your first analyzer and code fix

C# programming guide

Overview

Inside a C# program

What's inside a C# program

General Structure of a C# Program

Identifier names

C# Coding Conventions

Main() and command-line arguments

Overview

Command-Line Arguments

How to display command-line arguments

Main() Return Values

Top-level statements

Programming concepts

Overview

Asynchronous programming

Overview

Asynchronous programming scenarios

Task asynchronous programming model

Async return types

Cancel tasks

Cancel a list of tasks

Cancel tasks after a period of time

Process asynchronous tasks as they complete

Asynchronous file access

Attributes

Overview

Creating Custom Attributes

Accessing Attributes by Using Reflection

[How to create a C/C++ union by using attributes](#)

[Collections](#)

[Covariance and contravariance](#)

[Overview](#)

[Variance in Generic Interfaces](#)

[Create Variant Generic Interfaces](#)

[Use Variance in Interfaces for Generic Collections](#)

[Variance in Delegates](#)

[Use Variance in Delegates](#)

[Use Variance for Func and Action Generic Delegates](#)

[Expression trees](#)

[Overview](#)

[How to execute expression trees](#)

[How to modify expression trees](#)

[How to use expression trees to build dynamic queries](#)

[Debugging Expression Trees in Visual Studio](#)

[DebugView Syntax](#)

[Iterators](#)

[Language-Integrated Query \(LINQ\)](#)

[Overview](#)

[Getting Started with LINQ in C#](#)

[Introduction to LINQ Queries](#)

[LINQ and Generic Types](#)

[Basic LINQ Query Operations](#)

[Data Transformations with LINQ](#)

[Type Relationships in LINQ Query Operations](#)

[Query Syntax and Method Syntax in LINQ](#)

[C# Features That Support LINQ](#)

[Walkthrough: Writing Queries in C# \(LINQ\)](#)

[Standard Query Operators Overview](#)

[Overview](#)

[Query Expression Syntax for Standard Query Operators](#)

Classification of Standard Query Operators by Manner of Execution

[Sorting Data](#)

[Set Operations](#)

[Filtering Data](#)

[Quantifier Operations](#)

[Projection Operations](#)

[Partitioning Data](#)

[Join Operations](#)

[Grouping Data](#)

[Generation Operations](#)

[Equality Operations](#)

[Element Operations](#)

[Converting Data Types](#)

[Concatenation Operations](#)

[Aggregation Operations](#)

LINQ to Objects

[Overview](#)

[LINQ and Strings](#)

[How to articles](#)

[How to count occurrences of a word in a string \(LINQ\)](#)

[How to query for sentences that contain a specified set of words \(LINQ\)](#)

[How to query for characters in a string \(LINQ\)](#)

[How to combine LINQ queries with regular expressions](#)

[How to find the set difference between two lists \(LINQ\)](#)

[How to sort or filter text data by any word or field \(LINQ\)](#)

[How to reorder the fields of a delimited file \(LINQ\)](#)

[How to combine and compare string collections \(LINQ\)](#)

[How to populate object collections from multiple sources \(LINQ\)](#)

[How to split a file into many files by using groups \(LINQ\)](#)

[How to join content from dissimilar files \(LINQ\)](#)

[How to compute column values in a CSV text file \(LINQ\)](#)

LINQ and Reflection

[How to query an assembly's metadata with Reflection \(LINQ\)](#)

[LINQ and File Directories](#)

[Overview](#)

[How to query for files with a specified attribute or name](#)

[How to group files by extension \(LINQ\)](#)

[How to query for the total number of bytes in a set of folders \(LINQ\)](#)

[How to compare the contents of two folders \(LINQ\)](#)

[How to query for the largest file or files in a directory tree \(LINQ\)](#)

[How to query for duplicate files in a directory tree \(LINQ\)](#)

[How to query the contents of files in a folder \(LINQ\)](#)

[How to query an ArrayList with LINQ](#)

[How to add custom methods for LINQ queries](#)

[LINQ to ADO.NET \(Portal Page\)](#)

[Enabling a Data Source for LINQ Querying](#)

[Visual Studio IDE and Tools Support for LINQ](#)

[Reflection](#)

[Serialization \(C#\)](#)

[Overview](#)

[How to write object data to an XML file](#)

[How to read object data from an XML file](#)

[Walkthrough: Persisting an Object in Visual Studio](#)

[Statements, expressions, and operators](#)

[Overview](#)

[Statements](#)

[Expression-bodied members](#)

[Anonymous functions](#)

[Overview](#)

[How to use lambda expressions in a query](#)

[Equality and equality comparisons](#)

[Equality comparisons](#)

[How to define value equality for a type](#)

[How to test for reference equality \(identity\)](#)

Types

[Use and define types](#)

[Casting and Type Conversions](#)

[Boxing and Unboxing](#)

[How to convert a byte array to an int](#)

[How to convert a string to a number](#)

[How to convert between hexadecimal strings and numeric types](#)

[Using Type dynamic](#)

[Walkthrough: Creating and Using Dynamic Objects \(C# and Visual Basic\)](#)

Classes, Structs, and Records

[Overview](#)

[Classes](#)

[Records](#)

[Objects](#)

[Inheritance](#)

[Polymorphism](#)

[Overview](#)

[Versioning with the Override and New Keywords](#)

[Knowing When to Use Override and New Keywords](#)

[How to override the ToString method](#)

Members

[Members overview](#)

[Abstract and Sealed Classes and Class Members](#)

[Static Classes and Static Class Members](#)

[Access Modifiers](#)

[Fields](#)

[Constants](#)

[How to define abstract properties](#)

[How to define constants in C#](#)

Properties

[Properties overview](#)

[Using Properties](#)

Interface Properties

Restricting Accessor Accessibility

How to declare and use read write properties

Auto-Implemented Properties

How to implement a lightweight class with auto-implemented properties

Methods

Methods overview

Local functions

Ref returns and ref locals

Parameters

Passing parameters

Passing Value-Type Parameters

Passing Reference-Type Parameters

How to know the difference between passing a struct and passing a class reference to a method

Implicitly Typed Local Variables

How to use implicitly typed local variables and arrays in a query expression

Extension Methods

How to implement and call a custom extension method

How to create a new method for an enumeration

Named and Optional Arguments

How to use named and optional arguments in Office programming

Constructors

Constructors overview

Using Constructors

Instance Constructors

Private Constructors

Static Constructors

How to write a copy constructor

Finalizers

Object and Collection Initializers

How to initialize objects by using an object initializer

How to initialize a dictionary with a collection initializer

[Nested Types](#)

[Partial Classes and Methods](#)

[Anonymous Types](#)

[How to return subsets of element properties in a query](#)

[Interfaces](#)

[Overview](#)

[Explicit Interface Implementation](#)

[How to explicitly implement interface members](#)

[How to explicitly implement members of two interfaces](#)

[Delegates](#)

[Overview](#)

[Using Delegates](#)

[Delegates with Named vs. Anonymous Methods](#)

[How to combine delegates \(Multicast Delegates\) \(C# Programming Guide\)](#)

[How to declare, instantiate, and use a delegate](#)

[Arrays](#)

[Overview](#)

[Single-Dimensional Arrays](#)

[Multidimensional Arrays](#)

[Jagged Arrays](#)

[Using foreach with Arrays](#)

[Passing Arrays as Arguments](#)

[Implicitly Typed Arrays](#)

[Strings](#)

[Programming with strings](#)

[How to determine whether a string represents a numeric value](#)

[Indexers](#)

[Overview](#)

[Using Indexers](#)

[Indexers in Interfaces](#)

[Comparison Between Properties and Indexers](#)

[Events](#)

Overview

- How to subscribe to and unsubscribe from events
- How to publish events that conform to .NET Guidelines
- How to raise base class events in derived classes
- How to implement interface events
- How to implement custom event accessors

Generics

- Overview
- Generic Type Parameters
- Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generics and Arrays
- Generic Delegates
- Differences Between C++ Templates and C# Generics
- Generics in the Run Time
- Generics and Reflection
- Generics and Attributes

Namespaces

- Overview
- Using namespaces
- How to use the My namespace

XML documentation comments

- Overview
- Recommended tags for documentation comments
- Processing the XML file
- Delimiters for documentation tags
- How to use the XML documentation features
- Documentation tag reference
 - <c>
 - <code>

`cref` attribute

`<example>`

`<exception>`

`<include>`

`<inheritdoc>`

`<list>`

`<para>`

`<param>`

`<paramref>`

`<permission>`

`<remarks>`

`<returns>`

`<see>`

`<seealso>`

`<summary>`

`<typeparam>`

`<typeparamref>`

`<value>`

Exceptions and Exception Handling

[Overview](#)

[Using Exceptions](#)

[Exception Handling](#)

[Creating and Throwing Exceptions](#)

[Compiler-Generated Exceptions](#)

[How to handle an exception using try-catch](#)

[How to execute cleanup code using finally](#)

[How to catch a non-CLS exception](#)

File System and the Registry

[Overview](#)

[How to iterate through a directory tree](#)

[How to get information about files, folders, and drives](#)

[How to create a file or folder](#)

- How to copy, delete, and move files and folders
- How to provide a progress dialog box for file operations
- How to write to a text file
- How to read From a text file
- How to read a text file one line at a time
- How to create a key in the registry

Interoperability

- .NET Interoperability
- Interoperability Overview
- How to access Office interop objects by using C# features
- How to use indexed properties in COM interop programming
- How to use platform invoke to play a WAV file
- Walkthrough: Office Programming (C# and Visual Basic)
- Example COM Class

Language reference

Overview

Configure language version

Types

Value types

Overview

Integral numeric types

nint and nuint native integer types

Floating-point numeric types

Built-in numeric conversions

bool

char

Enumeration types

Structure types

Tuple types

Nullable value types

Reference types

Features of reference types

[Built-in reference types](#)

[record](#)

[class](#)

[interface](#)

[Nullable reference types](#)

[void](#)

[var](#)

[Built-in types](#)

[Unmanaged types](#)

[Default values](#)

[Keywords](#)

[Overview](#)

[Modifiers](#)

[Access Modifiers](#)

[Quick reference](#)

[Accessibility Levels](#)

[Accessibility Domain](#)

[Restrictions on Using Accessibility Levels](#)

[internal](#)

[private](#)

[protected](#)

[public](#)

[protected internal](#)

[private protected](#)

[abstract](#)

[async](#)

[const](#)

[event](#)

[extern](#)

[in \(generic modifier\)](#)

[new \(member modifier\)](#)

[out \(generic modifier\)](#)

`override`

`readonly`

`sealed`

`static`

`unsafe`

`virtual`

`volatile`

Statement Keywords

[Statement categories](#)

Selection Statements

`if-else`

`switch`

Iteration Statements

`do`

`for`

`foreach, in`

`while`

Jump Statements

`break`

`continue`

`goto`

`return`

Exception Handling Statements

`throw`

`try-catch`

`try-finally`

`try-catch-finally`

Checked and Unchecked

[Overview](#)

`checked`

`unchecked`

fixed Statement

[lock Statement](#)

[Method Parameters](#)

[Passing parameters](#)

[params](#)

[in \(Parameter Modifier\)](#)

[ref](#)

[out \(Parameter Modifier\)](#)

[Namespace Keywords](#)

[namespace](#)

[using](#)

[Contexts for using](#)

[using Directive](#)

[using static Directive](#)

[using Statement](#)

[extern alias](#)

[Type-testing Keywords](#)

[is](#)

[Generic Type Constraint Keywords](#)

[new constraint](#)

[where](#)

[Access Keywords](#)

[base](#)

[this](#)

[Literal Keywords](#)

[null](#)

[true and false](#)

[default](#)

[Contextual Keywords](#)

[Quick reference](#)

[add](#)

[get](#)

[init](#)

- partial (Type)
- partial (Method)
- remove
- set
- when (filter condition)
- value
- yield

Query Keywords

- Quick reference

- from clause

- where clause

- select clause

- group clause

- into

- orderby clause

- join clause

- let clause

- ascending

- descending

- on

- equals

- by

- in

Operators and expressions

- Overview

- Arithmetic operators

- Boolean logical operators

- Bitwise and shift operators

- Equality operators

- Comparison operators

- Member access operators and expressions

- Type-testing operators and cast expression

[User-defined conversion operators](#)

[Pointer-related operators](#)

[Assignment operators](#)

[Lambda expressions](#)

[Patterns](#)

[+ and += operators](#)

[- and -= operators](#)

[?: operator](#)

[! \(null-forgiving\) operator](#)

[?? and ??= operators](#)

[=> operator](#)

[:: operator](#)

[await operator](#)

[default value expressions](#)

[delegate operator](#)

[nameof expression](#)

[new operator](#)

[sizeof operator](#)

[stackalloc expression](#)

[switch expression](#)

[true and false operators](#)

[with expression](#)

[Operator overloading](#)

[Special characters](#)

[Overview](#)

[\\$ -- string interpolation](#)

[@ -- verbatim identifier](#)

[Attributes read by the compiler](#)

[Global attributes](#)

[Caller information](#)

[Nullable static analysis](#)

[Miscellaneous](#)

[Unsafe code and pointers](#)

[Preprocessor directives](#)

[Compiler options](#)

[How to set options](#)

[Language options](#)

[Output options](#)

[Input options](#)

[Error and warning options](#)

[Code generation options](#)

[Security options](#)

[Resources options](#)

[Miscellaneous options](#)

[Advanced options](#)

[Compiler errors](#)

[C# 6.0 draft specification](#)

[C# 7.0 - 9.0 proposals](#)

[Walkthroughs](#)

A tour of the C# language

3/6/2021 • 13 minutes to read • [Edit Online](#)

C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in the .NET ecosystem. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers. This tour provides an overview of the major components of the language in C# 8 and earlier. If you want to explore the language through interactive examples, try the [introduction to C#](#) tutorials.

C# is an object-oriented, **component-oriented** programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components. Since its origin, C# has added features to support new workloads and emerging software design practices.

Several C# features help create robust and durable applications. [Garbage collection](#) automatically reclaims memory occupied by unreachable unused objects. [Nullable types](#) guard against variables that don't refer to allocated objects. [Exception handling](#) provides a structured and extensible approach to error detection and recovery. [Lambda expressions](#) support functional programming techniques. [Language Integrated Query \(LINQ\)](#) syntax creates a common pattern for working with data from any source. Language support for [asynchronous operations](#) provides syntax for building distributed systems. C# has a [unified type system](#). All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. All types share a set of common operations. 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](#). C# allows dynamic allocation of objects and in-line storage of lightweight structures. C# supports generic methods and types, which provide increased type safety and performance. C# provides iterators, which enable implementers of collection classes to define custom behaviors for client code.

C# emphasizes **versioning** to ensure programs and libraries can evolve over time in a compatible manner. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

.NET architecture

C# programs run on .NET, a virtual execution system called the common language runtime (CLR) and a set of class libraries. The CLR is the implementation by Microsoft of the common language infrastructure (CLI), an international standard. The CLI is the basis for creating execution and development environments in which languages and libraries work together seamlessly.

Source code written in C# is compiled into an [intermediate language \(IL\)](#) that conforms to the CLI specification. The IL code and resources, such as bitmaps and strings, are stored in an assembly, typically with an extension of `.dll`. An assembly contains a manifest that provides information about the assembly's types, version, and culture.

When the C# program is executed, the assembly is loaded into the CLR. The CLR performs Just-In-Time (JIT) compilation to convert the IL code to native machine instructions. The CLR provides other services related to automatic garbage collection, exception handling, and resource management. Code that's executed by the CLR is sometimes referred to as "managed code," in contrast to "unmanaged code," which is compiled into native machine language that targets a specific platform.

Language interoperability is a key feature of .NET. IL code produced by the C# compiler conforms to the Common Type Specification (CTS). IL code generated from C# can interact with code that was generated from the .NET versions of F#, Visual Basic, C++, or any of more than 20 other CTS-compliant languages. A single

assembly may contain multiple modules written in different .NET languages, and the types can reference each other as if they were written in the same language.

In addition to the run time services, .NET also includes extensive libraries. These libraries support many different workloads. They're organized into namespaces that provide a wide variety of useful functionality for everything from file input and output to string manipulation to XML parsing, to web application frameworks to Windows Forms controls. The typical C# application uses the .NET class library extensively to handle common "plumbing" chores.

For more information about .NET, see [Overview of .NET](#).

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");
    }
}
```

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 C# 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 standard class libraries, which, by default, are automatically referenced by the compiler.

Types and variables

There are two kinds of types in C#: *value types* and *reference types*. Variables of value types directly contain their data. Variables of reference types store references to their data, the latter being known as objects. With reference types, it's possible for two variables to reference the same object and possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for `ref` and `out` parameter variables).

An *identifier* is a variable name. An identifier is a sequence of unicode characters without any whitespace. An identifier may be a C# reserved word, if it's prefixed by `@`. Using a reserved word as an identifier can be useful when interacting with other languages.

C#'s value types are further divided into *simple types*, *enum types*, *struct types*, *nullable value types*, and *tuple*

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

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

- **Value types**
 - **Simple types**
 - **Signed integral:** `sbyte`, `short`, `int`, `long`
 - **Unsigned integral:** `byte`, `ushort`, `uint`, `ulong`
 - **Unicode characters:** `char`, which represents a UTF-16 code unit
 - **IEEE binary floating-point:** `float`, `double`
 - **High-precision decimal floating-point:** `decimal`
 - **Boolean:** `bool`, which represents Boolean values—values that are either `true` or `false`
 - **Enum types**
 - User-defined types of the form `enum E { ... }`. An `enum` type is a distinct type with named constants. Every `enum` type has an underlying type, which must be one of the eight integral types. The set of values of an `enum` type is the same as the set of values of the underlying type.
 - **Struct types**
 - User-defined types of the form `struct S { ... }`
 - **Nullable value types**
 - Extensions of all other value types with a `null` value
 - **Tuple value types**
 - User-defined types of the form `(T1, T2, ...)`
- **Reference types**
 - **Class types**
 - Ultimate base class of all other types: `object`
 - **Unicode strings:** `string`, which represents a sequence of UTF-16 code units
 - User-defined types of the form `class C { ... }`
 - **Interface types**
 - User-defined types of the form `interface I { ... }`
 - **Array types**
 - Single-dimensional, multi-dimensional, and jagged. For example: `int[]`, `int[,]`, and `int[][]`
 - **Delegate types**
 - User-defined types of the form `delegate int D(...)`

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

- A `class` type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.
- A `struct` type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and don't typically require heap allocation. Struct types don't support user-specified inheritance, and all struct types implicitly inherit from type `object`.
- An `interface` type defines a contract as a named set of public members. A `class` or `struct` that implements an `interface` must provide implementations of the interface's members. An `interface` may inherit from multiple base interfaces, and a `class` or `struct` may implement multiple interfaces.
- A `delegate` type represents references to methods with a particular parameter list and return type.

Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They're also similar to the concept of function pointers found in some other languages. Unlike function pointers, delegates are object-oriented and type-safe.

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

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

Nullable types don't require a separate definition. For each non-nullable type `T`, there's 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`, and `string?` is a type that can hold any `string` or the value `null`.

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

```
int i = 123;
object o = i;      // Boxing
int j = (int)o;   // Unboxing
```

When a value of a value type is assigned to an `object` reference, a "box" is allocated to hold the value. That box is an instance of a reference type, 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. If the check succeeds, the value in the box is copied to the value type.

C#'s unified type system effectively means that value types are treated as `object` references "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with all types that derive from `object`, including both reference types and value types.

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

- 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 a compatible delegate type

Program structure

The key organizational concepts in C# are *programs*, *namespaces*, *types*, *members*, and *assemblies*.

Programs declare types, which contain members and can be organized into namespaces. Classes, structs, and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they're physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement *applications* or *libraries*, respectively.

As a small example, consider an assembly that contains the following code:

```
using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}
```

The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `_top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. The `Stack` is a *generic* class. It has one type parameter, `T` that is replaced with a concrete type when it's used.

NOTE

A *stack* is a "first in - last out" (FILO) collection. New elements are added to the top of the stack. When an element is removed, it is removed from the top of the stack.

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

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

```
using System;
using Acme.Collections;

class Example
{
    public static void Main()
    {
        var s = new Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

To compile this program, you would need to *reference* the assembly containing the stack class defined in the earlier example.

C# programs can be stored in several source files. When a C# program is compiled, all of the source files are processed together, and the source files can freely reference each other. Conceptually, it's as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with few exceptions, declaration order is insignificant. C# doesn't 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.

Further articles in this tour explain these organizational blocks.

NEXT

Types and members

3/6/2021 • 6 minutes to read • [Edit Online](#)

As an object-oriented language, C# supports the concepts of encapsulation, inheritance, and polymorphism. A class may inherit directly from one parent class, and it may implement any number of interfaces. Methods that override virtual methods in a parent class require the `override` keyword as a way to avoid accidental redefinition. In C#, a struct is like a lightweight class; it's a stack-allocated type that can implement interfaces but doesn't support inheritance. C# also provides records, which are class types whose purpose is primarily storing data values.

Classes and objects

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

New classes are created using class declarations. A class declaration starts with a header. The header specifies:

- The attributes and modifiers of the class
- The name of the class
- The base class (when inheriting from a [base class](#))
- 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 code shows a declaration of a simple class named `Point`:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

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

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

The memory occupied by an object is automatically reclaimed when the object is no longer reachable. It's not necessary or possible to explicitly deallocate objects in C#.

Type parameters

Generic classes define [type parameters](#). Type parameters are a list of type parameter names enclosed in angle brackets. Type parameters follow the class name. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of `Pair` are

`TFirst` and `TSecond`:

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

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

A class type that is declared to take type parameters is called a *generic class type*. Struct, interface, and delegate types can also be generic. When the generic class is used, type arguments must be provided for each of the type parameters:

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;      // TFirst int
string s = pair.Second; // TSecond 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. Follow 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`. From the first example, the base class of `Point` is `object`:

```
public class Point3D : Point
{
    public int Z { get; set; }

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

A class inherits the members of its base class. Inheritance means that a class implicitly contains almost all members of its base class. A class doesn't inherit the instance and static constructors, and the finalizer. A derived class can add new members to those members it inherits, but it can't remove the definition of an inherited member. In the previous example, `Point3D` inherits the `x` and `y` members from `Point`, and every `Point3D` instance contains three properties, `x`, `y`, and `z`.

An implicit conversion exists from a class type to any of its base class types. 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);
```

Structs

Classes define types that support inheritance and polymorphism. They enable you to create sophisticated behaviors based on hierarchies of derived classes. By contrast, *struct* types are simpler types whose primary purpose is to store data values. Structs can't declare a base type; they implicitly derive from `System.ValueType`.

You can't derive other `struct` types from a `struct` type. They're implicitly sealed.

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

Interfaces

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

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

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

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

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

interface IComboBox : ITextBox, IListBox { }
```

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

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

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

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

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

Enums

An [Enum](#) type defines a set of constant values. The following `enum` declares constants that define different root vegetables:

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

You can also define an `enum` to be used in combination as flags. The following declaration declares a set of flags for the four seasons. Any combination of the seasons may be applied, including an `All` value that includes all seasons:

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

The following example shows declarations of both the preceding enums:

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

Nullable types

Variables of any type may be declared as [*non-nullable*](#) or [*nullable*](#). A nullable variable can hold an additional `null` value, indicating no value. Nullable Value types (structs or enums) are represented by [System.Nullable<T>](#). Non-nullable and Nullable Reference types are both represented by the underlying reference type. The distinction is represented by metadata read by the compiler and some libraries. The compiler provides warnings when nullable references are dereferenced without first checking their value against `null`. The compiler also provides warnings when non-nullable references are assigned a value that may be `null`. The following example declares a [*nullable int*](#), initializing it to `null`. Then, it sets the value to `5`. It demonstrates the same concept with a [*nullable string*](#). For more information, see [nullable value types](#) and [nullable reference types](#).

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.;"
```

Tuples

C# supports [*tuples*](#), which provides concise syntax to group multiple data elements in a lightweight data

structure. You instantiate a tuple by declaring the types and names of the members between `(` and `)`, as shown in the following example:

```
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Tuples provide an alternative for data structure with multiple members, without using the building blocks described in the next article.

[PREVIOUS](#)

[NEXT](#)

Program building blocks

4/3/2021 • 23 minutes to read • [Edit Online](#)

The types described in the previous article are built using these building blocks: **members**, **expressions**, and **statements**.

Members

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

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

- **Constants**: Constant values associated with the class
- **Fields**: Variables that are associated with the class
- **Methods**: Actions that can be performed by the class
- **Properties**: Actions associated with reading and writing named properties of the class
- **Indexers**: Actions associated with indexing instances of the class like an array
- **Events**: Notifications that can be generated by the class
- **Operators**: Conversions and expression operators supported by the class
- **Constructors**: Actions required to initialize instances of the class or the class itself
- **Finalizers**: Actions performed before instances of the class are permanently discarded
- **Types**: Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that can access the member. There are six possible forms of accessibility. The access modifiers are summarized below.

- `public` : Access isn't limited.
- `private` : Access is limited to this class.
- `protected` : Access is limited to this class or classes derived from this class.
- `internal` : Access is limited to the current assembly (`.exe` or `.dll`).
- `protected internal` : Access is limited to this class, classes derived from this class, or classes within the same assembly.
- `private protected` : Access is limited to this class or classes derived from this type within the same assembly.

Fields

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

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

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

In the following example, each instance of the `Color` class has a separate copy of the `R`, `G`, and `B` instance fields, but there's only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

```

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

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
        B = b;
    }
}

```

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

Methods

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

Methods may have a list of *parameters*, which represent values or variable references passed to the method. Methods have a *return type*, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it doesn't return a value.

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

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

When a method body is a single expression, the method can be defined using a compact expression format, as shown in the following example:

```
public override string ToString() => "This is an object";
```

Parameters

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

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

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

A *reference parameter* is used for passing arguments by reference. The argument passed for a reference

parameter must be a variable with a definite value. During execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

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

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}"); // "2 1"
}
```

An *output parameter* is used for passing arguments by reference. It's similar to a reference parameter, except that it doesn't require that you explicitly assign a value to the caller-provided argument. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters using the syntax introduced in C# 7.

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

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}"); // "3 1"
}
```

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

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

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

```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

is equivalent to writing the following.

```
int x = 3, y = 4, z = 5;

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

Method body and local variables

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

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

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {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` didn't include an initial value, the compiler would report an error for the later usages of `i` because `i` wouldn't be definitely assigned at those points in the program.

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

Static and instance methods

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

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

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

```

class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

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

```

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

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

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

```

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"

```

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

Virtual, override, and abstract methods

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

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

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

An *abstract method* is a virtual method with no implementation. An abstract method is declared with the

`abstract` modifier and is permitted only in an abstract class. An abstract method must be overridden in every non-abstract derived class.

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

```

public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? 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)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> 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;

            default: throw new Exception("Unknown operator");
        }
    }
}

```

The previous four classes can be used to model arithmetic expressions. For example, using instances of these

classes, the expression `x + 3` can be represented as follows.

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

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

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

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

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new Dictionary<string, object>();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

Method overloading

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

```

class OverloadingExample
{
    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)");

    public static void UsageExample()
    {
        F();                  // Invokes F()
        F(1);                // Invokes F(int)
        F(1.0);              // Invokes F(double)
        F("abc");             // Invokes F<string>(string)
        F((double)1);         // Invokes F(double)
        F((object)1);         // Invokes F(object)
        F<int>(1);            // Invokes F<int>(int)
        F(1, 1);              // Invokes F(double, double)
    }
}

```

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

Other function members

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

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

```

public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get => _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;
            }
        }
    }

    public T this[int index]
    {
        get => _items[index];
        set
        {
            if (index < _count) _items[index] = value;
            else if (index == _count)
            {
                T[] newItems = new T[_count + 1];
                Array.Copy(_items, 0, newItems, 0, _count);
                newItems[_count] = value;
                _items = newItems;
            }
            else
            {
                T[] newItems = new T[_count + 1];
                Array.Copy(_items, 0, newItems, 0, index);
                newItems[index] = value;
                Array.Copy(_items, index, newItems, index + 1, _count - index);
                _items = newItems;
            }
        }
    }
}

```

```

public T this[int index]
{
    get => _items[index];
    set
    {
        _items[index] = value;
        OnChanged();
    }
}

public void Add(T item)
{
    if (_count == Capacity) Capacity = _count * 2;
    _items[_count] = item;
    _count++;
    OnChanged();
}
protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as MyList<T>);

static bool Equals(MyList<T> a, MyList<T> b)
{
    if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(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;
}

public event EventHandler Changed;

public static bool operator ==(MyList<T> a, MyList<T> b) =>
    Equals(a, b);

public static bool operator !=(MyList<T> a, MyList<T> b) =>
    !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's first loaded.

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

Instance constructors can be overloaded and can have optional parameters. For example, the `MyList<T>` class declares one instance constructor with a single optional `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `MyList<string>` instances using the constructor of the `MyList` class with and without the optional argument.

```

MyList<string> list1 = new MyList<string>();
MyList<string> list2 = new MyList<string>(10);

```

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

Properties

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

A property is declared like a field, except that the declaration ends with a get accessor 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*. A property that has only a set accessor is a *write-only property*.

A get accessor corresponds to a parameterless method with a return value of the property type. A set accessor corresponds to a method with a single parameter named value and no return type. The get accessor computes the value of the property. The set accessor provides a new value for the property. When the property is the target of an assignment, or the operand of `++` or `--`, the set accessor is invoked. In other cases where the property is referenced, the get accessor is invoked.

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

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

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

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

Indexers

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

The `MyList<T>` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `MyList<T>` instances with `int` values. For example:

```
MyList<string> names = new MyList<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. A class can declare multiple indexers as long as the number or types of their parameters differ.

Events

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

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

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

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

```
class EventExample
{
    static int s_changeCount;

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

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

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

Operators

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

The `MyList<T>` class declares two operators, `operator ==` and `operator !=`. These overridden operators give new meaning to expressions that apply those operators to `MyList` instances. Specifically, the operators define equality of two `MyList<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `MyList<int>` instances.

```
MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b); // Outputs "True"
b.Add(3);
Console.WriteLine(a == b); // Outputs "False"
```

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

Finalizers

A *finalizer* is a member that implements the actions required to finalize an instance of a class. Typically, a finalizer is needed to release unmanaged resources. Finalizers can't have parameters, they can't have accessibility modifiers, and they can't be invoked explicitly. The finalizer for an instance is invoked automatically during garbage collection. For more details, see the article on [finalizers](#).

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

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

Expressions

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

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

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

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

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

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

C# provides a number of operators to perform [arithmetic](#), [logical](#), [bitwise](#) and [shift](#) operations and [equality](#) and [order](#) comparisons.

For the complete list of C# operators ordered by precedence level, see [C# operators](#).

Statements

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

- A *block* permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.
- *Declaration statements* are used to declare local variables and constants.
- *Expression statements* are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound

assignment operators, increment and decrement operations using the `++` and `--` operators and `await` expressions.

- *Selection statements* are used to select one of a number of possible statements for execution based on the value of some expression. This group contains the `if` and `switch` statements.
- *Iteration statements* are used to execute repeatedly an embedded statement. This group contains the `while`, `do`, `for`, and `foreach` statements.
- *Jump statements* are used to transfer control. This group contains the `break`, `continue`, `goto`, `throw`, `return`, and `yield` statements.
- The `try ... catch` statement is used to catch exceptions that occur during execution of a block, and the `try ... finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.
- The `checked` and `unchecked` statements are used to control the overflow-checking context for integral-type arithmetic operations and conversions.
- The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.
- The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

The following lists the kinds of statements that can be used:

- Local variable declaration.
- Local constant declaration.
- Expression statement.
- `if` statement.
- `switch` statement.
- `while` statement.
- `do` statement.
- `for` statement.
- `foreach` statement.
- `break` statement.
- `continue` statement.
- `goto` statement.
- `return` statement.
- `yield` statement.
- `throw` statements and `try` statements.
- `checked` and `unchecked` statements.
- `lock` statement.
- `using` statement.

[PREVIOUS](#)

[NEXT](#)

Major language areas

3/25/2021 • 9 minutes to read • [Edit Online](#)

Arrays, collections, and LINQ

C# and .NET provide many different collection types. Arrays have syntax defined by the language. Generic collection types are listed in the [System.Collections.Generic](#) namespace. Specialized collections include [System.Span<T>](#) for accessing continuous memory on the stack frame, and [System.Memory<T>](#) for accessing continuous memory on the managed heap. All collections, including arrays, [Span<T>](#), and [Memory<T>](#) share a unifying principle for iteration. You use the [System.Collections.Generic.IEnumerable<T>](#) interface. This unifying principle means that any of the collection types can be used with LINQ queries or other algorithms. You write methods using [IEnumerable<T>](#) and those algorithms work with any collection.

Arrays

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

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

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

```
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[{i}] = {a[i]}");
}
```

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

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

The `a1` array contains 10 elements, the `a2` array contains 50 (10×5) elements, and the `a3` array contains 100 ($10 \times 5 \times 2$) elements. The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a [jagged array](#) because the lengths of the element arrays don't all have to be the same. The following example allocates an array of arrays of `int`:

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

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

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

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

The length of the array is inferred from the number of expressions between `{` and `}`. Array initialization can be shortened further such that the array type doesn't have to be restated.

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

Both of the previous examples are equivalent to the following code:

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

The `foreach` statement can be used to enumerate the elements of any collection. The following code enumerates the array from the preceding example:

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

The `foreach` statement uses the `IEnumerable<T>` interface, so can work with any collection.

String interpolation

C# *string interpolation* enables you to format strings by defining expressions whose results are placed in a format string. For example, the following example prints the temperature on a given day from a set of weather data:

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-DD-YYYY}");
Console.WriteLine($"    was {weatherData.LowTemp} and {weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//     was 5 and 30.
```

An interpolated string is declared using the `$` token. String interpolation evaluates the expressions between `{` and `}`, then converts the result to a `string`, and replaces the text between the brackets with the string result of the expression. The `:` in the first expression, `{weatherData.Date:MM-DD-YYYY}` specifies the *format string*. In the preceding example, it specifies that the date should be printed in "MM-DD-YYYY" format.

Pattern matching

The C# language provides **pattern matching** expressions to query the state of an object and execute code based on that state. You can inspect types and the values of properties and fields to determine which action to take. The `switch` expression is the primary expression for pattern matching.

Delegates and lambda expressions

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. Unlike function pointers, delegates are object-oriented and type-safe.

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

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}

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

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

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

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

Delegates can also be created using anonymous functions, which are "inline methods" that are created when declared. Anonymous functions can see the local variables of the surrounding methods. The following example doesn't create a class:

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

A delegate doesn't know or care about the class of the method it references. All that matters is that the referenced method has the same parameters and return type as the delegate.

async / await

C# supports asynchronous programs with two keywords: `async` and `await`. You add the `async` modifier to a method declaration to declare the method is asynchronous. The `await` operator tells the compiler to asynchronously await for a result to finish. Control is returned to the caller, and the method returns a structure that manages the state of the asynchronous work. The structure is typically `System.Threading.Tasks.Task<TResult>`, but can be any type that supports the awainer pattern. These features enable you to write code that reads as its synchronous counterpart, but executes asynchronously. For example, the following code downloads the home page for [Microsoft docs](#):

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished downloading.");
    return content.Length;
}
```

This small sample shows the major features for asynchronous programming:

- The method declaration includes the `async` modifier.
- The body of the method `awaits` the return of the `GetByteArrayAsync` method.
- The type specified in the `return` statement matches the type argument in the `Task<T>` declaration for the method. (A method that returns a `Task` would use `return` statements without any argument).

Attributes

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

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

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

All attribute classes derive from the [Attribute](#) base class provided by the .NET library. Attributes can be applied

by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` can be used as follows.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features",
        Topic = "Display")]
    public void Display(string text) { }
}
```

This example attaches a `HelpAttribute` to the `Widget` class. It adds another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The metadata defined by attributes can be read and manipulated at runtime using reflection. When a particular attribute is requested using this technique, the constructor for the attribute class is invoked with the information provided in the program source. 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.

The following code sample demonstrates how to get the `HelpAttribute` instances associated to the `Widget` class and its `Display` method.

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}
```

Learn more

You can explore more about C# by trying one of our [tutorials](#).

[PREVIOUS](#)

Introduction to C#

3/31/2021 • 2 minutes to read • [Edit Online](#)

Welcome to the introduction to C# tutorials. These lessons start with interactive code that you can run in your browser. You can learn the basics of C# from the [C# 101 video series](#) before starting these interactive lessons.

The first lessons explain C# concepts using small snippets of code. You'll learn the basics of C# syntax and how to work with data types like strings, numbers, and booleans. It's all interactive, and you'll be writing and running code within minutes. These first lessons assume no prior knowledge of programming or the C# language.

You can try these tutorials in different environments. The concepts you'll learn are the same. The difference is which experience you prefer:

- [In your browser, on the docs platform](#): This experience embeds a runnable C# code window in docs pages. You write and execute C# code in the browser.
- [In the Microsoft Learn experience](#): This learning path contains several modules that teach the basics of C#.
- [In Jupyter on Binder](#): You can experiment with C# code in a Jupyter notebook on binder.
- [On your local machine](#): After you've explored online, you can [download](#) the .NET Core SDK and build programs on your machine.

All the introductory tutorials following the Hello World lesson are available using the online browser experience or [in your own local development environment](#). At the end of each tutorial, you decide if you want to continue with the next lesson online or on your own machine. There are links to help you set up your environment and continue with the next tutorial on your machine.

Hello world

In the [Hello world](#) tutorial, you'll create the most basic C# program. You'll explore the `string` type and how to work with text. You can also use the path on [Microsoft Learn](#) or [Jupyter on Binder](#).

Numbers in C#

In the [Numbers in C#](#) tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding, and how to perform mathematical calculations using C#. This tutorial is also available [to run locally on your machine](#).

This tutorial assumes that you've finished the [Hello world](#) lesson.

Branches and loops

The [Branches and loops](#) tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions. This tutorial is also available [to run locally on your machine](#).

This tutorial assumes that you've finished the [Hello world](#) and [Numbers in C#](#) lessons.

List collection

The [List collection](#) lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists. This

tutorial is also available [to run locally on your machine](#).

This tutorial assumes that you've finished the lessons listed above.

101 Linq Samples

This sample requires the [dotnet-try](#) global tool. Once you install the tool, and clone the [try-samples](#) repo, you can learn Language Integrated Query (LINQ) through a set of 101 samples you can run interactively. You can explore different ways to query, explore, and transform data sequences.

Set up your local environment

3/6/2021 • 2 minutes to read • [Edit Online](#)

The first step in running a tutorial on your machine is to set up a development environment. Choose one of the following alternatives:

- To use the .NET CLI and your choice of text or code editor, see the .NET tutorial [Hello World in 10 minutes](#). The tutorial has instructions for setting up a development environment on Windows, Linux, or macOS.
- To use the .NET CLI and Visual Studio Code, install the [.NET SDK](#) and [Visual Studio Code](#).
- To use Visual Studio 2019, see [Tutorial: Create a simple C# console app in Visual Studio](#).

Basic application development flow

The instructions in these tutorials assume that you're using the .NET CLI to create, build, and run applications. You'll use the following commands:

- `dotnet new` creates an application. This command generates the files and assets necessary for your application. The introduction to C# tutorials all use the `console` application type. Once you've got the basics, you can expand to other application types.
- `dotnet build` builds the executable.
- `dotnet run` runs the executable.

If you use Visual Studio 2019 for these tutorials, you'll choose a Visual Studio menu selection when a tutorial directs you to run one of these CLI commands:

- **File > New > Project** creates an application.
- **Build > Build Solution** builds the executable.
- **Debug > Start Without Debugging** runs the executable.

Pick your tutorial

You can start with any of the following tutorials:

Numbers in C#

In the [Numbers in C#](#) tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding and how to perform mathematical calculations using C#.

This tutorial assumes that you have finished the [Hello world](#) lesson.

Branches and loops

The [Branches and loops](#) tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions.

This tutorial assumes that you have finished the [Hello world](#) and [Numbers in C#](#) lessons.

List collection

The [List collection](#) lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists.

This tutorial assumes that you have finished the lessons listed above.

Manipulate integral and floating point numbers in C#

3/23/2021 • 9 minutes to read • [Edit Online](#)

This tutorial teaches you about the numeric types in C# interactively. You'll write small amounts of code, then you'll compile and run that code. The tutorial contains a series of lessons that explore numbers and math operations in C#. These lessons teach you the fundamentals of the C# language.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Mac or Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

Explore integer math

Create a directory named *numbers-quickstart*. Make it the current directory and run the following command:

```
dotnet new console -n NumbersInCSharp -o .
```

Open *Program.cs* in your favorite editor, and replace the contents of the file with the following code:

```
using System;

int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You've seen one of the fundamental math operations with integers. The `int` type represents an **integer**, a zero, positive, or negative whole number. You use the `+` symbol for addition. Other common mathematical operations for integers include:

- `-` for subtraction
- `*` for multiplication
- `/` for division

Start by exploring those different operations. Add these lines after the line that writes the value of `c`:

```
// subtraction  
c = a - b;  
Console.WriteLine(c);  
  
// multiplication  
c = a * b;  
Console.WriteLine(c);  
  
// division  
c = a / b;  
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You can also experiment by writing multiple mathematics operations in the same line, if you'd like. Try

```
c = a + b - 12 * 17;
```

for example. Mixing variables and constant numbers is allowed.

TIP

As you explore C# (or any programming language), you'll make mistakes when you write code. The **compiler** will find those errors and report them to you. When the contains error messages, look closely at the example code and the code in your window to see what to fix. That exercise will help you learn the structure of C# code.

You've finished the first step. Before you start the next section, let's move the current code into a separate *method*. A method is a series of statements grouped together and given a name. You call a method by writing the method's name followed by `()`. Organizing your code into methods makes it easier to start working with a new example. When you finish, your code should look like this:

```
using System;  
  
WorkWithIntegers();  
  
void WorkWithIntegers()  
{  
    int a = 18;  
    int b = 6;  
    int c = a + b;  
    Console.WriteLine(c);  
  
    // subtraction  
    c = a - b;  
    Console.WriteLine(c);  
  
    // multiplication  
    c = a * b;  
    Console.WriteLine(c);  
  
    // division  
    c = a / b;  
    Console.WriteLine(c);  
}
```

The line `WorkWithIntegers();` invokes the method. The following code declares the method and defines it.

Explore order of operations

Comment out the call to `WorkingWithIntegers()`. It will make the output less cluttered as you work in this section:

```
//WorkWithIntegers();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler doesn't generate any executable code from comments. Because `WorkWithIntegers()` is a method, you need to only comment out one line.

The C# language defines the precedence of different mathematics operations with rules consistent with the rules you learned in mathematics. Multiplication and division take precedence over addition and subtraction. Explore that by adding the following code after the call to `WorkWithIntegers()`, and executing `dotnet run`:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

The output demonstrates that the multiplication is performed before the addition.

You can force a different order of operation by adding parentheses around the operation or operations you want performed first. Add the following lines and run again:

```
d = (a + b) * c;
Console.WriteLine(d);
```

Explore more by combining many different operations. Add something like the following lines. Try `dotnet run` again.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

You may have noticed an interesting behavior for integers. Integer division always produces an integer result, even when you'd expect the result to include a decimal or fractional portion.

If you haven't seen this behavior, try the following code:

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Type `dotnet run` again to see the results.

Before moving on, let's take all the code you've written in this section and put it in a new method. Call that new method `OrderPrecedence`. Your code should look something like this:

```
using System;
```

```
// WorkWithIntegers();
```

```
OrderPrecedence();
```

```
void WorkWithIntegers()
```

```
{
```

```
    int a = 18;
```

```
    int b = 6;
```

```
    int c = a + b;
```

```
    Console.WriteLine(c);
```

```
// subtraction
```

```
c = a - b;
```

```
Console.WriteLine(c);
```

```
// multiplication
```

```
c = a * b;
```

```
Console.WriteLine(c);
```

```
// division
```

```
c = a / b;
```

```
Console.WriteLine(c);
```

```
}
```

```
void OrderPrecedence()
```

```
{
```

```
    int a = 5;
```

```
    int b = 4;
```

```
    int c = 2;
```

```
    int d = a + b * c;
```

```
    Console.WriteLine(d);
```

```
d = (a + b) * c;
```

```
Console.WriteLine(d);
```

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
```

```
Console.WriteLine(d);
```

```
int e = 7;
```

```
int f = 4;
```

```
int g = 3;
```

```
int h = (e + f) / g;
```

```
Console.WriteLine(h);
```

```
}
```

Explore integer precision and limits

That last sample showed you that integer division truncates the result. You can get the **remainder** by using the **modulo** operator, the `%` character. Try the following code after the method call to `OrderPrecedence()`:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

The C# integer type differs from mathematical integers in one other way: the `int` type has minimum and maximum limits. Add this code to see those limits:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

If a calculation produces a value that exceeds those limits, you have an **underflow** or **overflow** condition. The answer appears to wrap from one limit to the other. Add these two lines to see an example:

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notice that the answer is very close to the minimum (negative) integer. It's the same as `min + 2`. The addition operation **overflows** the allowed values for integers. The answer is a very large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

There are other numeric types with different limits and precision that you would use when the `int` type doesn't meet your needs. Let's explore those other types next. Before you start the next section, move the code you wrote in this section into a separate method. Name it `TestLimits`.

Work with the double type

The `double` numeric type represents a double-precision floating point number. Those terms may be new to you. A **floating point number** is useful to represent non-integral numbers that may be very large or small in magnitude. **Double-precision** is a relative term that describes the number of binary digits used to store the value. **Double precision** numbers have twice the number of binary digits as **single-precision**. On modern computers, it's more common to use double precision than single precision numbers. **Single precision** numbers are declared using the `float` keyword. Let's explore. Add the following code and see the result:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Notice that the answer includes the decimal portion of the quotient. Try a slightly more complicated expression with doubles:

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

The range of a double value is much greater than integer values. Try the following code below what you've written so far:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

These values are printed in scientific notation. The number to the left of the `E` is the significand. The number to the right is the exponent, as a power of 10. Just like decimal numbers in math, doubles in C# can have rounding errors. Try this code:

```
double third = 1.0 / 3.0;  
Console.WriteLine(third);
```

You know that `0.3` repeating isn't exactly the same as `1/3`.

Challenge

Try other calculations with large numbers, small numbers, multiplication, and division using the `double` type.

Try more complicated calculations. After you've spent some time with the challenge, take the code you've written and place it in a new method. Name that new method `WorkWithDoubles`.

Work with decimal types

You've seen the basic numeric types in C#: integers and doubles. There's one other type to learn: the `decimal` type. The `decimal` type has a smaller range but greater precision than `double`. Let's take a look:

```
decimal min = decimal.MinValue;  
decimal max = decimal.MaxValue;  
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notice that the range is smaller than the `double` type. You can see the greater precision with the decimal type by trying the following code:

```
double a = 1.0;  
double b = 3.0;  
Console.WriteLine(a / b);  
  
decimal c = 1.0M;  
decimal d = 3.0M;  
Console.WriteLine(c / d);
```

The `M` suffix on the numbers is how you indicate that a constant should use the `decimal` type. Otherwise, the compiler assumes the `double` type.

NOTE

The letter `M` was chosen as the most visually distinct letter between the `double` and `decimal` keywords.

Notice that the math using the decimal type has more digits to the right of the decimal point.

Challenge

Now that you've seen the different numeric types, write code that calculates the area of a circle whose radius is 2.50 centimeters. Remember that the area of a circle is the radius squared multiplied by PI. One hint: .NET contains a constant for PI, `Math.PI` that you can use for that value. `Math.PI`, like all constants declared in the `System.Math` namespace, is a `double` value. For that reason, you should use `double` instead of `decimal` values for this challenge.

You should get an answer between 19 and 20. You can check your answer by [looking at the finished sample code on GitHub](#).

Try some other formulas if you'd like.

You've completed the "Numbers in C#" quickstart. You can continue with the [Branches and loops](#) quickstart in your own development environment.

You can learn more about numbers in C# in the following articles:

- [Integral numeric types](#)
- [Floating-point numeric types](#)
- [Built-in numeric conversions](#)

Learn conditional logic with branch and loop statements

3/23/2021 • 9 minutes to read • [Edit Online](#)

This tutorial teaches you how to write code that examines variables and changes the execution path based on those variables. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that explore branching and looping constructs in C#. These lessons teach you the fundamentals of the C# language.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Mac and Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

Make decisions using the `if` statement

Create a directory named *branches-tutorial*. Make that the current directory and run the following command:

```
dotnet new console -n BranchesAndLoops -o .
```

This command creates a new .NET console application in the current directory. Open *Program.cs* in your favorite editor, and replace the contents with the following code:

```
using System;

int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Try this code by typing `dotnet run` in your console window. You should see the message "The answer is greater than 10." printed to your console. Modify the declaration of `b` so that the sum is less than 10:

```
int b = 3;
```

Type `dotnet run` again. Because the answer is less than 10, nothing is printed. The **condition** you're testing is false. You don't have any code to execute because you've only written one of the possible branches for an `if` statement: the true branch.

TIP

As you explore C# (or any programming language), you'll make mistakes when you write code. The compiler will find and report the errors. Look closely at the error output and the code that generated the error. The compiler error can usually help you find the problem.

This first sample shows the power of `if` and Boolean types. A *Boolean* is a variable that can have one of two

values: `true` or `false`. C# defines a special type, `bool` for Boolean variables. The `if` statement checks the value of a `bool`. When the value is `true`, the statement following the `if` executes. Otherwise, it's skipped. This process of checking conditions and executing statements based on those conditions is powerful.

Make if and else work together

To execute different code in both the true and false branches, you create an `else` branch that executes when the condition is false. Try an `else` branch. Add the last two lines in the code below (you should already have the first four):

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

The statement following the `else` keyword executes only when the condition being tested is `false`. Combining `if` and `else` with Boolean conditions provides all the power you need to handle both a `true` and a `false` condition.

IMPORTANT

The indentation under the `if` and `else` statements is for human readers. The C# language doesn't treat indentation or white space as significant. The statement following the `if` or `else` keyword will be executed based on the condition. All the samples in this tutorial follow a common practice to indent lines based on the control flow of statements.

Because indentation isn't significant, you need to use `{` and `}` to indicate when you want more than one statement to be part of the block that executes conditionally. C# programmers typically use those braces on all `if` and `else` clauses. The following example is the same as the one you created. Modify your code above to match the following code:

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

TIP

Through the rest of this tutorial, the code samples all include the braces, following accepted practices.

You can test more complicated conditions. Add the following code after the code you've written so far:

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

The `==` symbol tests for *equality*. Using `==` distinguishes the test for equality from assignment, which you saw in `a = 5`.

The `&&` represents "and". It means both conditions must be true to execute the statement in the true branch. These examples also show that you can have multiple statements in each conditional branch, provided you enclose them in `{` and `}`. You can also use `||` to represent "or". Add the following code after what you've written so far:

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modify the values of `a`, `b`, and `c` and switch between `&&` and `||` to explore. You'll gain more understanding of how the `&&` and `||` operators work.

You've finished the first step. Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Put the existing code in a method called `ExploreIf()`. Call it from the top of your program. When you finished those changes, your code should look like the following:

```

using System;

ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the second");
    }
}

```

Comment out the call to `ExploreIf()`. It will make the output less cluttered as you work in this section:

```
//ExploreIf();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler doesn't generate any executable code from comments.

Use loops to repeat operations

In this section, you use **loops** to repeat statements. Add this code after the call to `ExploreIf`:

```

int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}

```

The `while` statement checks a condition and executes the statement or statement block following the `while`. It repeatedly checks the condition and executing those statements until the condition is false.

There's one other new operator in this example. The `++` after the `counter` variable is the **increment** operator. It adds 1 to the value of `counter` and stores that value in the `counter` variable.

IMPORTANT

Make sure that the `while` loop condition changes to false as you execute the code. Otherwise, you create an **infinite loop** where your program never ends. That is not demonstrated in this sample, because you have to force your program to quit using **CTRL-C** or other means.

The `while` loop tests the condition before executing the code following the `while`. The `do ... while` loop executes the code first, and then checks the condition. The *do while* loop is shown in the following code:

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

This `do` loop and the earlier `while` loop produce the same output.

Work with the `for` loop

The `for` loop is commonly used in C#. Try this code:

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

The previous code does the same work as the `while` loop and the `do` loop you've already used. The `for` statement has three parts that control how it works.

The first part is the **for initializer**: `int index = 0;` declares that `index` is the loop variable, and sets its initial value to `0`.

The middle part is the **for condition**: `index < 10` declares that this `for` loop continues to execute as long as the value of counter is less than 10.

The final part is the **for iterator**: `index++` specifies how to modify the loop variable after executing the block following the `for` statement. Here, it specifies that `index` should be incremented by 1 each time the block executes.

Experiment yourself. Try each of the following variations:

- Change the initializer to start at a different value.
- Change the condition to stop at a different value.

When you're done, let's move on to write some code yourself to use what you've learned.

There's one other looping statement that isn't covered in this tutorial: the `foreach` statement. The `foreach` statement repeats its statement for every item in a sequence of items. It's most often used with *collections*, so it's covered in the next tutorial.

Created nested loops

A `while`, `do`, or `for` loop can be nested inside another loop to create a matrix using the combination of each item in the outer loop with each item in the inner loop. Let's do that to build a set of alphanumeric pairs to represent rows and columns.

One `for` loop can generate the rows:

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

Another loop can generate the columns:

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

You can nest one loop inside the other to form pairs:

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

You can see that the outer loop increments once for each full run of the inner loop. Reverse the row and column nesting, and see the changes for yourself. When you're done, place the code from this section in a method called `ExploreLoops()`.

Combine branches and loops

Now that you've seen the `if` statement and the looping constructs in the C# language, see if you can write C# code to find the sum of all integers 1 through 20 that are divisible by 3. Here are a few hints:

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.

Try it yourself. Then check how you did. You should get 63 for an answer. You can see one possible answer by [viewing the completed code on GitHub](#).

You've completed the "branches and loops" tutorial.

You can continue with the [Arrays and collections](#) tutorial in your own development environment.

You can learn more about these concepts in these articles:

- [If and else statement](#)
- [While statement](#)
- [Do statement](#)
- [For statement](#)

Learn to manage data collections using the generic list type

3/25/2021 • 5 minutes to read • [Edit Online](#)

This introductory tutorial provides an introduction to the C# language and the basics of the `List<T>` class.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Mac and Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

A basic list example

Create a directory named `list-tutorial`. Make that the current directory and run `dotnet new console`.

Open `Program.cs` in your favorite editor, and replace the existing code with the following:

```
using System;
using System.Collections.Generic;

var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Replace `<name>` with your name. Save `Program.cs`. Type `dotnet run` in your console window to try it.

You've created a list of strings, added three names to that list, and printed the names in all CAPS. You're using concepts that you've learned in earlier tutorials to loop through the list.

The code to display names makes use of the [string interpolation](#) feature. When you precede a `string` with the `$` character, you can embed C# code in the string declaration. The actual string replaces that C# code with the value it generates. In this example, it replaces the `{name.ToUpper()}` with each name, converted to capital letters, because you called the `ToUpper` method.

Let's keep exploring.

Modify list contents

The collection you created uses the `List<T>` type. This type stores sequences of elements. You specify the type of the elements between the angle brackets.

One important aspect of this `List<T>` type is that it can grow or shrink, enabling you to add or remove elements. Add this code at the end of your program:

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

You've added two more names to the end of the list. You've also removed one as well. Save the file, and type `dotnet run` to try it.

The `List<T>` enables you to reference individual items by `index` as well. You place the index between `[` and `]` tokens following the list name. C# uses 0 for the first index. Add this code directly below the code you just added and try it:

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

You can't access an index beyond the end of the list. Remember that indices start at 0, so the largest valid index is one less than the number of items in the list. You can check how long the list is using the `Count` property. Add the following code at the end of your program:

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Save the file, and type `dotnet run` again to see the results.

Search and sort lists

Our samples use relatively small lists, but your applications may often create lists with many more elements, sometimes numbering in the thousands. To find elements in these larger collections, you need to search the list for different items. The `IndexOf` method searches for an item and returns the index of the item. If the item isn't in the list, `IndexOf` returns `-1`. Add this code to the bottom of your program:

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}
```

The items in your list can be sorted as well. The `Sort` method sorts all the items in the list in their normal order (alphabetically for strings). Add this code to the bottom of your program:

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Save the file and type `dotnet run` to try this latest version.

Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Place all the code you've written in a new method called `WorkWithStrings()`. Call that method at the top of your program. When you finish, your code should look like this:

```

using System;
using System.Collections.Generic;

WorkWithString();

void WorkWithString()
{
    var names = new List<string> { "<name>", "Ana", "Felipe" };
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine();
    names.Add("Maria");
    names.Add("Bill");
    names.Remove("Ana");
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine($"My name is {names[0]}");
    Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

    Console.WriteLine($"The list has {names.Count} people in it");

    var index = names.IndexOf("Felipe");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns {index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    index = names.IndexOf("Not Found");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns {index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    names.Sort();
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }
}

```

Lists of other types

You've been using the `string` type in lists so far. Let's make a `List<T>` using a different type. Let's build a set of numbers.

Add the following to your program after you call `WorkWithStrings()`:

```
var fibonacciNumbers = new List<int> {1, 1};
```

That creates a list of integers, and sets the first two integers to the value 1. These are the first two values of a *Fibonacci Sequence*, a sequence of numbers. Each next Fibonacci number is found by taking the sum of the previous two numbers. Add this code:

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Save the file and type `dotnet run` to see the results.

TIP

To concentrate on just this section, you can comment out the code that calls `WorkingWithStrings();`. Just put two `/` characters in front of the call like this: `// WorkingWithStrings();`.

Challenge

See if you can put together some of the concepts from this and earlier lessons. Expand on what you've built so far with Fibonacci Numbers. Try to write the code to generate the first 20 numbers in the sequence. (As a hint, the 20th Fibonacci number is 6765.)

Complete challenge

You can see an example solution by [looking at the finished sample code on GitHub](#).

With each iteration of the loop, you're taking the last two integers in the list, summing them, and adding that value to the list. The loop repeats until you've added 20 items to the list.

Congratulations, you've completed the list tutorial. You can continue with [additional](#) tutorials in your own development environment.

You can learn more about working with the `List` type in the .NET fundamentals article on [collections](#). You'll also learn about many other collection types.

What's new in C# 9.0

3/23/2021 • 16 minutes to read • [Edit Online](#)

C# 9.0 adds the following features and enhancements to the C# language:

- [Records](#)
- [Init only setters](#)
- [Top-level statements](#)
- [Pattern matching enhancements](#)
- Native sized integers
- Function pointers
- Suppress emitting localsinit flag
- Target-typed new expressions
- static anonymous functions
- Target-typed conditional expressions
- Covariant return types
- Extension `GetEnumerator` support for `foreach` loops
- Lambda discard parameters
- Attributes on local functions
- Module initializers
- New features for partial methods

C# 9.0 is supported on .NET 5. For more information, see [C# language versioning](#).

You can download the latest .NET SDK from the [.NET downloads page](#).

Record types

C# 9.0 introduces *record types*, which are a reference type that provides synthesized methods to provide value semantics for equality. Records are immutable by default.

Record types make it easy to create immutable reference types in .NET. Historically, .NET types are largely classified as reference types (including classes and anonymous types) and value types (including structs and tuples). While immutable value types are recommended, mutable value types don't often introduce errors. Value type variables hold the values so changes are made to a copy of the original data when value types are passed to methods.

There are many advantages to immutable reference types as well. These advantages are more pronounced in concurrent programs with shared data. Unfortunately, C# forced you to write quite a bit of extra code to create immutable reference types. Records provide a type declaration for an immutable reference type that uses value semantics for equality. The synthesized methods for equality and hash codes consider two records equal if their properties are all equal. Consider this definition:

```

public record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last) => (FirstName, LastName) = (first, last);
}

```

The record definition creates a `Person` type that contains two readonly properties: `FirstName` and `LastName`.

The `Person` type is a reference type. If you looked at the IL, it's a class. It's immutable in that none of the properties can be modified once it's been created. When you define a record type, the compiler synthesizes several other methods for you:

- Methods for value-based equality comparisons
- Override for `GetHashCode()`
- Copy and Clone members
- `PrintMembers` and `ToString()`

Records support inheritance. You can declare a new record derived from `Person` as follows:

```

public record Teacher : Person
{
    public string Subject { get; }

    public Teacher(string first, string last, string sub)
        : base(first, last) => Subject = sub;
}

```

You can also seal records to prevent further derivation:

```

public sealed record Student : Person
{
    public int Level { get; }

    public Student(string first, string last, int level) : base(first, last) => Level = level;
}

```

The compiler synthesizes different versions of the methods above. The method signatures depend on if the record type is sealed and if the direct base class is object. Records should have the following capabilities:

- Equality is value-based, and includes a check that the types match. For example, a `Student` can't be equal to a `Person`, even if the two records share the same name.
- Records have a consistent string representation generated for you.
- Records support copy construction. Correct copy construction must include inheritance hierarchies, and properties added by developers.
- Records can be copied with modification. These copy and modify operations supports non-destructive mutation.

In addition to the familiar `Equals` overloads, `operator ==`, and `operator !=`, the compiler synthesizes a new `EqualityContract` property. The property returns a `Type` object that matches the type of the record. If the base type is `object`, the property is `virtual`. If the base type is another record type, the property is an `override`. If the record type is `sealed`, the property is `sealed`. The synthesized `GetHashCode` uses the `GetHashCode` from all properties and fields declared in the base type and the record type. These synthesized methods enforce value-based equality throughout an inheritance hierarchy. That means a `Student` will never be considered equal to a `Person` with the same name. The types of the two records must match as well as all properties shared among

the record types being equal.

Records also have a synthesized constructor and a "clone" method for creating copies. The synthesized constructor has a single parameter of the record type. It produces a new record with the same values for all properties of the record. This constructor is private if the record is sealed, otherwise it's protected. The synthesized "clone" method supports copy construction for record hierarchies. The term "clone" is in quotes because the actual name is compiler generated. You can't create a method named `Clone` in a record type. The synthesized "clone" method returns the type of record being copied using virtual dispatch. The compiler adds different modifiers for the "clone" method depending on the access modifiers on the `record`:

- If the record type is `abstract`, the "clone" method is also `abstract`. If the base type isn't `object`, the method is also `override`.
- For record types that aren't `abstract` when the base type is `object`:
 - If the record is `sealed`, no additional modifiers are added to the "clone" method (meaning it is not `virtual`).
 - If the record isn't `sealed`, the "clone" method is `virtual`.
- For record types that aren't `abstract` when the base type is not `object`:
 - If the record is `sealed`, the "clone" method is also `sealed`.
 - If the record isn't `sealed`, the "clone" method is `override`.

The result of all these rules is the equality is implemented consistently across any hierarchy of record types. Two records are equal to each other if their properties are equal and their types are the same, as shown in the following example:

```
var person = new Person("Bill", "Wagner");
var student = new Student("Bill", "Wagner", 11);

Console.WriteLine(student == person); // false
```

The compiler synthesizes two methods that support printed output: a `ToString()` override, and `PrintMembers`. The `PrintMembers` takes a `System.Text.StringBuilder` as its argument. It appends a comma-separated list of property names and values for all properties in the record type. `PrintMembers` calls the base implementation for any records derived from other records. The `ToString()` override returns the string produced by `PrintMembers`, surrounded by `{` and `}`. For example, the `ToString()` method for `Student` returns a `string` like the following code:

```
"Student { LastName = Wagner, FirstName = Bill, Level = 11 }"
```

The examples shown so far use traditional syntax to declare properties. There's a more concise form called **positional records**. Here are the three record types defined earlier as positional records:

```
public record Person(string FirstName, string LastName);

public record Teacher(string FirstName, string LastName,
    string Subject)
: Person(FirstName, LastName);

public sealed record Student(string FirstName,
    string LastName, int Level)
: Person(FirstName, LastName);
```

These declarations create the same functionality as the earlier version (with a couple extra features covered in the following section). These declarations end with a semicolon instead of brackets because these records don't

add additional methods. You can add a body, and include any additional methods as well:

```
public record Pet(string Name)
{
    public void ShredTheFurniture() =>
        Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
    public void WagTail() =>
        Console.WriteLine("It's tail wagging time");

    public override string ToString()
    {
        StringBuilder s = new();
        base.PrintMembers(s);
        return $"{s.ToString()} is a dog";
    }
}
```

The compiler produces a `Deconstruct` method for positional records. The `Deconstruct` method has parameters that match the names of all public properties in the record type. The `Deconstruct` method can be used to deconstruct the record into its component properties:

```
var person = new Person("Bill", "Wagner");

var (first, last) = person;
Console.WriteLine(first);
Console.WriteLine(last);
```

Finally, records support `with` expressions. A `with` expression instructs the compiler to create a copy of a record, but `with` specified properties modified:

```
Person brother = person with { FirstName = "Paul" };
```

The previous line creates a new `Person` record where the `LastName` property is a copy of `person`, and the `FirstName` is `"Paul"`. You can set any number of properties in a `with` expression. You can also use `with` expressions to create an exact copy. You specify the empty set for the properties to modify:

```
Person clone = person with { };
```

Any of the synthesized members except the "clone" method may be written by you. If a record type has a method that matches the signature of any synthesized method, the compiler doesn't synthesize that method. The earlier `Dog` record example contains a hand coded `ToString()` method as an example.

Learn more about record types in this [exploration of records](#) tutorial.

Init only setters

Init only setters provide consistent syntax to initialize members of an object. Property initializers make it clear which value is setting which property. The downside is that those properties must be settable. Starting with C# 9.0, you can create `init` accessors instead of `set` accessors for properties and indexers. Callers can use property initializer syntax to set these values in creation expressions, but those properties are readonly once construction has completed. Init only setters provide a window to change state. That window closes when the construction phase ends. The construction phase effectively ends after all initialization, including property

initializers and with-expressions have completed.

You can declare `init` only setters in any type you write. For example, the following struct defines a weather observation structure:

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

Callers can use property initializer syntax to set the values, while still preserving the immutability:

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

But, changing an observation after initialization is an error by assigning to an init-only property outside of initialization:

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Init only setters can be useful to set base class properties from derived classes. They can also set derived properties through helpers in a base class. Positional records declare properties using init only setters. Those setters are used in with-expressions. You can declare init only setters for any `class` or `struct` you define.

Top-level statements

Top-level statements remove unnecessary ceremony from many applications. Consider the canonical "Hello World!" program:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

There's only one line of code that does anything. With top-level statements, you can replace all that boilerplate with the `using` statement and the single line that does the work:

```
using System;

Console.WriteLine("Hello World!");
```

If you wanted a one-line program, you could remove the `using` directive and use the fully qualified type name:

```
System.Console.WriteLine("Hello World!");
```

Only one file in your application may use top-level statements. If the compiler finds top-level statements in multiple source files, it's an error. It's also an error if you combine top-level statements with a declared program entry point method, typically a `Main` method. In a sense, you can think that one file contains the statements that would normally be in the `Main` method of a `Program` class.

One of the most common uses for this feature is creating teaching materials. Beginner C# developers can write the canonical "Hello World!" in one or two lines of code. None of the extra ceremony is needed. However, seasoned developers will find many uses for this feature as well. Top-level statements enable a script-like experience for experimentation similar to what Jupyter notebooks provide. Top-level statements are great for small console programs and utilities. Azure Functions are an ideal use case for top-level statements.

Most importantly, top-level statements don't limit your application's scope or complexity. Those statements can access or use any .NET class. They also don't limit your use of command-line arguments or return values. Top-level statements can access an array of strings named `args`. If the top-level statements return an integer value, that value becomes the integer return code from a synthesized `Main` method. The top-level statements may contain `async` expressions. In that case, the synthesized entry point returns a `Task`, or `Task<int>`.

Pattern matching enhancements

C# 9 includes new pattern matching improvements:

- *Type patterns* match a variable is a type
- *Parenthesized patterns* enforce or emphasize the precedence of pattern combinations
- *Conjunctive* `and` *patterns* require both patterns to match
- *Disjunctive* `or` *patterns* require either pattern to match
- *Negated* `not` *patterns* require that a pattern doesn't match
- *Relational patterns* require the input be less than, greater than, less than or equal, or greater than or equal to a given constant.

These patterns enrich the syntax for patterns. Consider these examples:

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Alternatively, with optional parentheses to make it clear that `and` has higher precedence than `or`:

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

One of the most common uses is a new syntax for a null check:

```
if (e is not null)
{
    // ...
}
```

Any of these patterns can be used in any context where patterns are allowed: `is` pattern expressions, `switch` expressions, nested patterns, and the pattern of a `switch` statement's `case` label.

Performance and interop

Three new features improve support for native interop and low-level libraries that require high performance: native sized integers, function pointers, and omitting the `localsinit` flag.

Native sized integers, `nint` and `nuint`, are integer types. They're expressed by the underlying types `System.IntPtr` and `System.UIntPtr`. The compiler surfaces additional conversions and operations for these types as native ints. Native sized integers define properties for `MaxValue` or `MinValue`. These values can't be expressed as compile time constants because they depend on the native size of an integer on the target machine. Those values are readonly at runtime. You can use constant values for `nint` in the range [`int.MinValue .. int.MaxValue`]. You can use constant values for `nuint` in the range [`uint.MinValue .. uint.MaxValue`]. The compiler performs constant folding for all unary and binary operators using the `System.Int32` and `System.UInt32` types. If the result doesn't fit in 32 bits, the operation is executed at runtime and isn't considered a constant. Native sized integers can increase performance in scenarios where integer math is used extensively and needs to have the fastest performance possible.

Function pointers provide an easy syntax to access the IL opcodes `ldftn` and `calli`. You can declare function pointers using new `delegate*` syntax. A `delegate*` type is a pointer type. Invoking the `delegate*` type uses `calli`, in contrast to a delegate that uses `callvirt` on the `Invoke()` method. Syntactically, the invocations are identical. Function pointer invocation uses the `managed` calling convention. You add the `unmanaged` keyword after the `delegate*` syntax to declare that you want the `unmanaged` calling convention. Other calling conventions can be specified using attributes on the `delegate*` declaration.

Finally, you can add the `System.Runtime.CompilerServices.SkipLocalsInitAttribute` to instruct the compiler not to emit the `localsinit` flag. This flag instructs the CLR to zero-initialize all local variables. The `localsinit` flag has been the default behavior for C# since 1.0. However, the extra zero-initialization may have measurable performance impact in some scenarios. In particular, when you use `stackalloc`. In those cases, you can add the `SkipLocalsInitAttribute`. You may add it to a single method or property, or to a `class`, `struct`, `interface`, or even a module. This attribute doesn't affect `abstract` methods; it affects the code generated for the implementation.

These features can improve performance in some scenarios. They should be used only after careful benchmarking both before and after adoption. Code involving native sized integers must be tested on multiple target platforms with different integer sizes. The other features require unsafe code.

Fit and finish features

Many of the other features help you write code more efficiently. In C# 9.0, you can omit the type in a `new` expression when the created object's type is already known. The most common use is in field declarations:

```
private List<WeatherObservation> _observations = new();
```

Target-typed `new` can also be used when you need to create a new object to pass as an argument to a method. Consider a `ForecastFor()` method with the following signature:

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

You could call it as follows:

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

Another nice use for this feature is to combine it with init only properties to initialize a new object:

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

You can return an instance created by the default constructor using a `return new();` statement.

A similar feature improves the target type resolution of [conditional expressions](#). With this change, the two expressions need not have an implicit conversion from one to the other, but may both have implicit conversions to a target type. You likely won't notice this change. What you will notice is that some conditional expressions that previously required casts or wouldn't compile now just work.

Starting in C# 9.0, you can add the `static` modifier to [lambda expressions](#) or [anonymous methods](#). Static lambda expressions are analogous to the `static` local functions: a static lambda or anonymous method can't capture local variables or instance state. The `static` modifier prevents accidentally capturing other variables.

Covariant return types provide flexibility for the return types of [override](#) methods. An override method can return a type derived from the return type of the overridden base method. This can be useful for records and for other types that support virtual clone or factory methods.

In addition, the `foreach` loop will recognize and use an extension method `GetEnumerator` that otherwise satisfies the `foreach` pattern. This change means `foreach` is consistent with other pattern-based constructions such as the `async` pattern, and pattern-based deconstruction. In practice, this change means you can add `foreach` support to any type. You should limit its use to when enumerating an object makes sense in your design.

Next, you can use discards as parameters to lambda expressions. This convenience enables you to avoid naming the argument, and the compiler may avoid using it. You use the `_` for any argument. For more information, see the [Input parameters of a lambda expression](#) section of the [Lambda expressions](#) article.

Finally, you can now apply attributes to [local functions](#). For example, you can apply [nullable attribute annotations](#) to local functions.

Support for code generators

Two final features support C# code generators. C# code generators are a component you can write that is similar to a roslyn analyzer or code fix. The difference is that code generators analyze code and write new source code files as part of the compilation process. A typical code generator searches code for attributes or other conventions.

A code generator reads attributes or other code elements using the Roslyn analysis APIs. From that information, it adds new code to the compilation. Source generators can only add code; they aren't allowed to modify any existing code in the compilation.

The two features added for code generators are extensions to [partial method syntax](#), and [module initializers](#). First, the changes to partial methods. Before C# 9.0, partial methods are `private` but can't specify an access modifier, have a `void` return, and can't have `out` parameters. These restrictions meant that if no method implementation is provided, the compiler removes all calls to the partial method. C# 9.0 removes these restrictions, but requires that partial method declarations have an implementation. Code generators can provide

that implementation. To avoid introducing a breaking change, the compiler considers any partial method without an access modifier to follow the old rules. If the partial method includes the `private` access modifier, the new rules govern that partial method.

The second new feature for code generators is *module initializers*. Module initializers are methods that have the [ModuleInitializerAttribute](#) attribute attached to them. These methods will be called by the runtime before any other field access or method invocation within the entire module. A module initializer method:

- Must be static
- Must be parameterless
- Must return void
- Must not be a generic method
- Must not be contained in a generic class
- Must be accessible from the containing module

That last bullet point effectively means the method and its containing class must be internal or public. The method can't be a local function.

What's new in C# 8.0

3/23/2021 • 16 minutes to read • [Edit Online](#)

C# 8.0 adds the following features and enhancements to the C# language:

- [Readonly members](#)
- [Default interface methods](#)
- [Pattern matching enhancements:](#)
 - [Switch expressions](#)
 - [Property patterns](#)
 - [Tuple patterns](#)
 - [Positional patterns](#)
- [Using declarations](#)
- [Static local functions](#)
- [Disposable ref structs](#)
- [Nullable reference types](#)
- [Asynchronous streams](#)
- [Asynchronous disposable](#)
- [Indices and ranges](#)
- [Null-coalescing assignment](#)
- [Unmanaged constructed types](#)
- [Stackalloc in nested expressions](#)
- [Enhancement of interpolated verbatim strings](#)

C# 8.0 is supported on .NET Core 3.x and .NET Standard 2.1. For more information, see [C# language versioning](#).

The remainder of this article briefly describes these features. Where in-depth articles are available, links to those tutorials and overviews are provided. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the [dotnet-try](#) global tool.
2. Clone the [dotnet/try-samples](#) repository.
3. Set the current directory to the `csharp8` subdirectory for the `try-samples` repository.
4. Run `dotnet try`.

Readonly members

You can apply the `readonly` modifier to members of a struct. It indicates that the member doesn't modify state.

It's more granular than applying the `readonly` modifier to a `struct` declaration. Consider the following mutable struct:

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"{{X}}, {{Y}} is {Distance} from the origin";
}
```

Like most structs, the `ToString()` method doesn't modify state. You could indicate that by adding the `readonly` modifier to the declaration of `ToString()`:

```
public readonly override string ToString() =>
    $"{{X}}, {{Y}} is {Distance} from the origin";
```

The preceding change generates a compiler warning, because `ToString` accesses the `Distance` property, which isn't marked `readonly`:

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an implicit copy of 'this'
```

The compiler warns you when it needs to create a defensive copy. The `Distance` property doesn't change state, so you can fix this warning by adding the `readonly` modifier to the declaration:

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Notice that the `readonly` modifier is necessary on a read-only property. The compiler doesn't assume `get` accessors don't modify state; you must declare `readonly` explicitly. Auto-implemented properties are an exception; the compiler will treat all auto-implemented getters as `readonly`, so here there's no need to add the `readonly` modifier to the `X` and `Y` properties.

The compiler does enforce the rule that `readonly` members don't modify state. The following method won't compile unless you remove the `readonly` modifier:

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

This feature lets you specify your design intent so the compiler can enforce it, and make optimizations based on that intent.

For more information, see the [readonly instance members](#) section of the [Structure types](#) article.

Default interface methods

You can now add members to interfaces and provide an implementation for those members. This language feature enables API authors to add methods to an interface in later versions without breaking source or binary compatibility with existing implementations of that interface. Existing implementations *inherit* the default implementation. This feature also enables C# to interoperate with APIs that target Android or Swift, which support similar features. Default interface methods also enable scenarios similar to a "traits" language feature.

Default interface methods affect many scenarios and language elements. Our first tutorial covers [updating an interface with default implementations](#).

More patterns in more places

Pattern matching gives tools to provide shape-dependent functionality across related but different kinds of data. C# 7.0 introduced syntax for type patterns and constant patterns by using the `is` expression and the `switch` statement. These features represented the first tentative steps toward supporting programming paradigms where data and functionality live apart. As the industry moves toward more microservices and other cloud-based architectures, other language tools are needed.

C# 8.0 expands this vocabulary so you can use more pattern expressions in more places in your code. Consider these features when your data and functionality are separate. Consider pattern matching when your algorithms depend on a fact other than the runtime type of an object. These techniques provide another way to express designs.

In addition to new patterns in new places, C# 8.0 adds **recursive patterns**. The result of any pattern expression is an expression. A recursive pattern is simply a pattern expression applied to the output of another pattern expression.

Switch expressions

Often, a `switch` statement produces a value in each of its `case` blocks. **Switch expressions** enable you to use more concise expression syntax. There are fewer repetitive `case` and `break` keywords, and fewer curly braces. As an example, consider the following enum that lists the colors of the rainbow:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

If your application defined an `RGBColor` type that is constructed from the `R`, `G` and `B` components, you could convert a `Rainbow` value to its RGB values using the following method containing a switch expression:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red      => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange   => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow   => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green    => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue     => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo   => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet   => new RGBColor(0x94, 0x00, 0xD3),
        _                  => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
}
```

There are several syntax improvements here:

- The variable comes before the `switch` keyword. The different order makes it visually easy to distinguish the switch expression from the switch statement.

- The `case` and `:` elements are replaced with `=>`. It's more concise and intuitive.
- The `default` case is replaced with a `_` discard.
- The bodies are expressions, not statements.

Contrast that with the equivalent code using the classic `switch` statement:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    }
}
```

Property patterns

The **property pattern** enables you to match on properties of the object examined. Consider an eCommerce site that must compute sales tax based on the buyer's address. That computation isn't a core responsibility of an `Address` class. It will change over time, likely more often than address format changes. The amount of sales tax depends on the `State` property of the address. The following method uses the property pattern to compute the sales tax from the address and the price:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
}
```

Pattern matching creates a concise syntax for expressing this algorithm.

Tuple patterns

Some algorithms depend on multiple inputs. **Tuple patterns** allow you to switch based on multiple values expressed as a `tuple`. The following code shows a switch expression for the game *rock, paper, scissors*.

```

public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };

```

The messages indicate the winner. The discard case represents the three combinations for ties, or other text inputs.

Positional patterns

Some types include a `Deconstruct` method that deconstructs its properties into discrete variables. When a `Deconstruct` method is accessible, you can use **positional patterns** to inspect properties of the object and use those properties for a pattern. Consider the following `Point` class that includes a `Deconstruct` method to create discrete variables for `x` and `y`:

```

public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}

```

Additionally, consider the following enum that represents various positions of a quadrant:

```

public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}

```

The following method uses the **positional pattern** to extract the values of `x` and `y`. Then, it uses a `when` clause to determine the `Quadrant` of the point:

```

static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};

```

The discard pattern in the preceding switch matches when either `x` or `y` is 0, but not both. A switch expression must either produce a value or throw an exception. If none of the cases match, the switch expression throws an exception. The compiler generates a warning for you if you don't cover all possible cases in your switch expression.

You can explore pattern matching techniques in this [advanced tutorial on pattern matching](#).

Using declarations

A **using declaration** is a variable declaration preceded by the `using` keyword. It tells the compiler that the variable being declared should be disposed at the end of the enclosing scope. For example, consider the following code that writes a text file:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

In the preceding example, the file is disposed when the closing brace for the method is reached. That's the end of the scope in which `file` is declared. The preceding code is equivalent to the following code that uses the classic [using statement](#):

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

In the preceding example, the file is disposed when the closing brace associated with the `using` statement is reached.

In both cases, the compiler generates the call to `Dispose()`. The compiler generates an error if the expression in

the `using` statement isn't disposable.

Static local functions

You can now add the `static` modifier to [local functions](#) to ensure that local function doesn't capture (reference) any variables from the enclosing scope. Doing so generates [CS8421](#), "A static local function can't contain a reference to <variable>."

Consider the following code. The local function `LocalFunction` accesses the variable `y`, declared in the enclosing scope (the method `M`). Therefore, `LocalFunction` can't be declared with the `static` modifier:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

The following code contains a static local function. It can be static because it doesn't access any variables in the enclosing scope:

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

Disposable ref structs

A `struct` declared with the `ref` modifier may not implement any interfaces and so can't implement [IDisposable](#). Therefore, to enable a `ref struct` to be disposed, it must have an accessible `void Dispose()` method. This feature also applies to `readonly ref struct` declarations.

Nullable reference types

Inside a nullable annotation context, any variable of a reference type is considered to be a **nonnullable reference type**. If you want to indicate that a variable may be null, you must append the type name with the `?` to declare the variable as a **nullable reference type**.

For nonnullable reference types, the compiler uses flow analysis to ensure that local variables are initialized to a non-null value when declared. Fields must be initialized during construction. The compiler generates a warning if the variable isn't set by a call to any of the available constructors or by an initializer. Furthermore, nonnullable reference types can't be assigned a value that could be null.

Nullable reference types aren't checked to ensure they aren't assigned or initialized to null. However, the compiler uses flow analysis to ensure that any variable of a nullable reference type is checked against null before it's accessed or assigned to a nonnullable reference type.

You can learn more about the feature in the overview of [nullable reference types](#). Try it yourself in a new application in this [nullable reference types tutorial](#). Learn about the steps to migrate an existing codebase to make use of nullable reference types in the [migrating an application to use nullable reference types tutorial](#).

Asynchronous streams

Starting with C# 8.0, you can create and consume streams asynchronously. A method that returns an asynchronous stream has three properties:

1. It's declared with the `async` modifier.
2. It returns an `IAsyncEnumerable<T>`.
3. The method contains `yield return` statements to return successive elements in the asynchronous stream.

Consuming an asynchronous stream requires you to add the `await` keyword before the `foreach` keyword when you enumerate the elements of the stream. Adding the `await` keyword requires the method that enumerates the asynchronous stream to be declared with the `async` modifier and to return a type allowed for an `async` method. Typically that means returning a `Task` or `Task<TResult>`. It can also be a `ValueTask` or `ValueTask<TResult>`. A method can both consume and produce an asynchronous stream, which means it would return an `IAsyncEnumerable<T>`. The following code generates a sequence from 0 to 19, waiting 100 ms between generating each number:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

You would enumerate the sequence using the `await foreach` statement:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

You can try asynchronous streams yourself in our tutorial on [creating and consuming async streams](#). By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the `TaskAsyncEnumerableExtensions.ConfigureAwait` extension method. For more information about synchronization contexts and capturing the current context, see the article on [consuming the Task-based asynchronous pattern](#).

Asynchronous disposable

Starting with C# 8.0, the language supports asynchronous disposable types that implement the `System.IAsyncDisposable` interface. You use the `await using` statement to work with an asynchronously disposable object. For more information, see the [Implement a DisposeAsync method](#) article.

Indices and ranges

Indices and ranges provide a succinct syntax for accessing single elements or ranges in a sequence.

This language support relies on two new types, and two new operators:

- `System.Index` represents an index into a sequence.
- The index from end operator `^`, which specifies that an index is relative to the end of the sequence.
- `System.Range` represents a sub range of a sequence.
- The range operator `...`, which specifies the start and end of a range as its operands.

Let's start with the rules for indexes. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. Note that `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence.Length - n`.

A range specifies the *start* and *end* of a range. The start of the range is inclusive, but the end of the range is exclusive, meaning the *start* is included in the range but the *end* isn't included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

Let's look at a few examples. Consider the following array, annotated with its index from the start and from the end:

```
var words = new string[]
{
    "The",           // 0           ^9
    "quick",         // 1           ^8
    "brown",         // 2           ^7
    "fox",           // 3           ^6
    "jumped",        // 4           ^5
    "over",          // 5           ^4
    "the",           // 6           ^3
    "lazy",          // 7           ^2
    "dog"            // 8           ^1
};

// 9 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index:

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range.

```
var quickBrownFox = words[1..4];
```

The following code creates a subrange with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included:

```
var lazyDog = words[^2..^0];
```

The following examples create ranges that are open ended for the start, end, or both:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

You can also declare ranges as variables:

```
Range phrase = 1..4;
```

The range can then be used inside the `[` and `]` characters:

```
var text = words[phrase];
```

Not only arrays support indices and ranges. You can also use indices and ranges with [string](#), [Span<T>](#), or [ReadOnlySpan<T>](#). For more information, see [Type support for indices and ranges](#).

You can explore more about indices and ranges in the tutorial on [indices and ranges](#).

Null-coalescing assignment

C# 8.0 introduces the null-coalescing assignment operator `??=`. You can use the `??=` operator to assign the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`.

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers)); // output: 17 17
Console.WriteLine(i); // output: 17
```

For more information, see the [?? and ??= operators](#) article.

Unmanaged constructed types

In C# 7.3 and earlier, a constructed type (a type that includes at least one type argument) can't be an [unmanaged type](#). Starting with C# 8.0, a constructed value type is unmanaged if it contains fields of unmanaged types only.

For example, given the following definition of the generic `Coords<T>` type:

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

the `Coords<int>` type is an unmanaged type in C# 8.0 and later. Like for any unmanaged type, you can create a pointer to a variable of this type or [allocate a block of memory on the stack](#) for instances of this type:

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

For more information, see [Unmanaged types](#).

Stackalloc in nested expressions

Starting with C# 8.0, if the result of a `stackalloc` expression is of the [System.Span<T>](#) or [System.ReadOnlySpan<T>](#) type, you can use the `stackalloc` expression in other expressions:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

Enhancement of interpolated verbatim strings

Order of the `$` and `@` tokens in [interpolated](#) verbatim strings can be any: both `$$@"..."` and `@$"..."` are valid interpolated verbatim strings. In earlier C# versions, the `$` token must appear before the `@` token.

What's new in C# 7.0 through C# 7.3

3/23/2021 • 24 minutes to read • [Edit Online](#)

C# 7.0 through C# 7.3 brought a number of features and incremental improvements to your development experience with C#. This article provides an overview of the new language features and compiler options. The descriptions describe the behavior for C# 7.3, which is the most recent version supported for .NET Framework-based applications.

The [language version selection](#) configuration element was added with C# 7.1, which enables you to specify the compiler language version in your project file.

C# 7.0-7.3 adds these features and themes to the C# language:

- [Tuples and discards](#)
 - You can create lightweight, unnamed types that contain multiple public fields. Compilers and IDE tools understand the semantics of these types.
 - Discards are temporary, write-only variables used in assignments when you don't care about the value assigned. They're most useful when deconstructing tuples and user-defined types, as well as when calling methods with `out` parameters.
- [Pattern Matching](#)
 - You can create branching logic based on arbitrary types and values of the members of those types.
- [async Main method](#)
 - The entry point for an application can have the `async` modifier.
- [Local Functions](#)
 - You can nest functions inside other functions to limit their scope and visibility.
- [More expression-bodied members](#)
 - The list of members that can be authored using expressions has grown.
- [throw Expressions](#)
 - You can throw exceptions in code constructs that previously weren't allowed because `throw` was a statement.
- [default literal expressions](#)
 - You can use default literal expressions in default value expressions when the target type can be inferred.
- [Numeric literal syntax improvements](#)
 - New tokens improve readability for numeric constants.
- [out variables](#)
 - You can declare `out` values inline as arguments to the method where they're used.
- [Non-trailing named arguments](#)
 - Named arguments can be followed by positional arguments.
- [private protected access modifier](#)
 - The `private protected` access modifier enables access for derived classes in the same assembly.
- [Improved overload resolution](#)
 - New rules to resolve overload resolution ambiguity.
- [Techniques for writing safe efficient code](#)
 - A combination of syntax improvements that enable working with value types using reference semantics.

Finally, the compiler has new options:

- `-refout` and `-refonly` that control [reference assembly generation](#).
- `-publicsign` to enable Open Source Software (OSS) signing of assemblies.
- `-pathmap` to provide a mapping for source directories.

The remainder of this article provides an overview of each feature. For each feature, you'll learn the reasoning behind it and the syntax. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the [dotnet-try](#) global tool.
2. Clone the [dotnet/try-samples](#) repository.
3. Set the current directory to the `csharp7` subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Tuples and discards

C# provides a rich syntax for classes and structs that is used to explain your design intent. But sometimes that rich syntax requires extra work with minimal benefit. You may often write methods that need a simple structure containing more than one data element. To support these scenarios *tuples* were added to C#. Tuples are lightweight data structures that contain multiple fields to represent the data members. The fields aren't validated, and you can't define your own methods. C# tuple types support `==` and `!=`. For more information.

NOTE

Tuples were available before C# 7.0, but they were inefficient and had no language support. This meant that tuple elements could only be referenced as `Item1`, `Item2` and so on. C# 7.0 introduces language support for tuples, which enables semantic names for the fields of a tuple using new, more efficient tuple types.

You can create a tuple by assigning a value to each member, and optionally providing semantic names to each of the members of the tuple:

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

The `namedLetters` tuple contains fields referred to as `Alpha` and `Beta`. Those names exist only at compile time and aren't preserved, for example when inspecting the tuple using reflection at run time.

In a tuple assignment, you can also specify the names of the fields on the right-hand side of the assignment:

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

There may be times when you want to unpack the members of a tuple that were returned from a method. You can do that by declaring separate variables for each of the values in the tuple. This unpackaging is called *deconstructing* the tuple:

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

You can also provide a similar deconstruction for any type in .NET. You write a `Deconstruct` method as a member of the class. That `Deconstruct` method provides a set of `out` arguments for each of the properties you

want to extract. Consider this `Point` class that provides a deconstructor method that extracts the `x` and `y` coordinates:

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

You can extract the individual fields by assigning a `Point` to a tuple:

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

Many times when you initialize a tuple, the variables used for the right side of the assignment are the same as the names you'd like for the tuple elements: The names of tuple elements can be inferred from the variables used to initialize the tuple:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

You can learn more about this feature in the [Tuple types](#) article.

Often when deconstructing a tuple or calling a method with `out` parameters, you're forced to define a variable whose value you don't care about and don't intend to use. C# adds support for *discards* to handle this scenario. A discard is a write-only variable whose name is `_` (the underscore character); you can assign all of the values that you intend to discard to the single variable. A discard is like an unassigned variable; apart from the assignment statement, the discard can't be used in code.

Discards are supported in the following scenarios:

- When deconstructing tuples or user-defined types.
- When calling methods with `out` parameters.
- In a pattern matching operation with the `is` and `switch` statements.
- As a standalone identifier when you want to explicitly identify the value of an assignment as a discard.

The following example defines a `QueryCityDataForYears` method that returns a 6-tuple that contains data for a city for two different years. The method call in the example is concerned only with the two population values returned by the method and so treats the remaining values in the tuple as discards when it deconstructs the tuple.

```

using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}

// The example displays the following output:
//      Population change, 1960 to 2010: 393,149

```

For more information, see [Discards](#).

Pattern matching

Pattern matching is a set of features that enable new ways to express control flow in your code. You can test variables for their type, values or the values of their properties. These techniques create more readable code flow.

Pattern matching supports `is` expressions and `switch` expressions. Each enables inspecting an object and its properties to determine if that object satisfies the sought pattern. You use the `when` keyword to specify additional rules to the pattern.

The `is` pattern expression extends the familiar `is` operator to query an object about its type and assign the result in one instruction. The following code checks if a variable is an `int`, and if so, adds it to the current sum:

```

if (input is int count)
    sum += count;

```

The preceding small example demonstrates the enhancements to the `is` expression. You can test against value types as well as reference types, and you can assign the successful result to a new variable of the correct type.

The switch match expression has a familiar syntax, based on the `switch` statement already part of the C# language. The updated switch statement has several new constructs:

- The governing type of a `switch` expression is no longer restricted to integral types, `Enum` types, `string`, or a nullable type corresponding to one of those types. Any type may be used.
- You can test the type of the `switch` expression in each `case` label. As with the `is` expression, you may assign a new variable to that type.
- You may add a `when` clause to further test conditions on that variable.
- The order of `case` labels is now important. The first branch to match is executed; others are skipped.

The following code demonstrates these new features:

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` is the familiar constant pattern.
- `case IEnumerable<int> childSequence:` is a type pattern.
- `case int n when n > 0:` is a type pattern with an additional `when` condition.
- `case null:` is the null pattern.
- `default:` is the familiar default case.

Beginning with C# 7.1, the pattern expression for `is` and the `switch` type pattern may have the type of a generic type parameter. This can be most useful when checking types that may be either `struct` or `class` types, and you want to avoid boxing.

You can learn more about pattern matching in [Pattern Matching in C#](#).

Async main

An `async main` method enables you to use `await` in your `Main` method. Previously you would need to write:

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

You can now write:

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

If your program doesn't return an exit code, you can declare a `Main` method that returns a `Task`:

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

You can read more about the details in the [async main](#) article in the programming guide.

Local functions

Many designs for classes include methods that are called from only one location. These additional private methods keep each method small and focused. *Local functions* enable you to declare methods inside the context of another method. Local functions make it easier for readers of the class to see that the local method is only called from the context in which it is declared.

There are two common use cases for local functions: public iterator methods and public async methods. Both types of methods generate code that reports errors later than programmers might expect. In iterator methods, any exceptions are observed only when calling code that enumerates the returned sequence. In async methods, any exceptions are only observed when the returned `Task` is awaited. The following example demonstrates separating parameter validation from the iterator implementation using a local function:

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than {nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

The same technique can be employed with `async` methods to ensure that exceptions arising from argument validation are thrown before the asynchronous work begins:

```

public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}

```

This syntax is now supported:

```

[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }

```

The attribute `SomeThingAboutFieldAttribute` is applied to the compiler generated backing field for `SomeProperty`. For more information, see [attributes](#) in the C# programming guide.

NOTE

Some of the designs that are supported by local functions can also be accomplished using *lambda expressions*. For more information, see [Local functions vs. lambda expressions](#).

More expression-bodied members

C# 6 introduced expression-bodied members for member functions and read-only properties. C# 7.0 expands the allowed members that can be implemented as expressions. In C# 7.0, you can implement *constructors*, *finalizers*, and `get` and `set` accessors on *properties* and *indexers*. The following code shows examples of each:

```

// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}

```

NOTE

This example does not need a finalizer, but it is shown to demonstrate the syntax. You should not implement a finalizer in your class unless it is necessary to release unmanaged resources. You should also consider using the [SafeHandle](#) class instead of managing unmanaged resources directly.

These new locations for expression-bodied members represent an important milestone for the C# language: These features were implemented by community members working on the open-source [Roslyn](#) project.

Changing a method to an expression bodied member is a [binary compatible change](#).

Throw expressions

In C#, `throw` has always been a statement. Because `throw` is a statement, not an expression, there were C# constructs where you couldn't use it. These included conditional expressions, null coalescing expressions, and some lambda expressions. The addition of expression-bodied members adds more locations where `throw` expressions would be useful. So that you can write any of these constructs, C# 7.0 introduces [throw expressions](#).

This addition makes it easier to write more expression-based code. You don't need additional statements for error checking.

Default literal expressions

Default literal expressions are an enhancement to default value expressions. These expressions initialize a variable to the default value. Where you previously would write:

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

You can now omit the type on the right-hand side of the initialization:

```
Func<string, bool> whereClause = default;
```

For more information, see the [default literal](#) section of the [default operator](#) article.

Numeric literal syntax improvements

Misreading numeric constants can make it harder to understand code when reading it for the first time. Bit masks or other symbolic values are prone to misunderstanding. C# 7.0 includes two new features to write numbers in the most readable fashion for the intended use: *binary literals*, and *digit separators*.

For those times when you're creating bit masks, or whenever a binary representation of a number makes the most readable code, write that number in binary:

```
public const int Sixteen = 0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

The `0b` at the beginning of the constant indicates that the number is written as a binary number. Binary numbers can get long, so it's often easier to see the bit patterns by introducing the `_` as a digit separator, as shown in the binary constant in the preceding example. The digit separator can appear anywhere in the constant. For base 10 numbers, it is common to use it as a thousands separator. Hex and binary numeric literals may begin with an `_`:

```
public const long BillionsAndBillions = 100_000_000_000;
```

The digit separator can be used with `decimal`, `float`, and `double` types as well:

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

Taken together, you can declare numeric constants with much more readability.

out variables

The existing syntax that supports `out` parameters has been improved in C# 7. You can now declare `out` variables in the argument list of a method call, rather than writing a separate declaration statement:

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

You may want to specify the type of the `out` variable for clarity, as shown in the preceding example. However, the language does support using an implicitly typed local variable:

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- The code is easier to read.
 - You declare the `out` variable where you use it, not on a preceding line of code.
- No need to assign an initial value.
 - By declaring the `out` variable where it's used in a method call, you can't accidentally use it before it is assigned.

The syntax added in C# 7.0 to allow `out` variable declarations has been extended to include field initializers, property initializers, constructor initializers, and query clauses. It enables code such as the following example:

```
public class B
{
    public B(int i, out int j)
    {
        j = i;
    }
}

public class D : B
{
    public D(int i) : base(i, out var j)
    {
        Console.WriteLine($"The value of 'j' is {j}");
    }
}
```

Non-trailing named arguments

Method calls may now use named arguments that precede positional arguments when those named arguments are in the correct positions. For more information, see [Named and optional arguments](#).

private protected access modifier

A new compound access modifier: `private protected` indicates that a member may be accessed by containing class or derived classes that are declared in the same assembly. While `protected internal` allows access by derived classes or classes that are in the same assembly, `private protected` limits access to derived types declared in the same assembly.

For more information, see [access modifiers](#) in the language reference.

Improved overload candidates

In every release, the overload resolution rules get updated to address situations where ambiguous method invocations have an "obvious" choice. This release adds three new rules to help the compiler pick the obvious choice:

1. When a method group contains both instance and static members, the compiler discards the instance members if the method was invoked without an instance receiver or context. The compiler discards the static members if the method was invoked with an instance receiver. When there is no receiver, the compiler includes only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type, the compiler includes both. A static context, where an implicit `this` instance receiver cannot be used, includes the body of members where no `this` is defined, such as static members, as well as places where `this` cannot be used, such as field initializers and constructor-initializers.
2. When a method group contains some generic methods whose type arguments do not satisfy their constraints, these members are removed from the candidate set.
3. For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

You'll only notice this change because you'll find fewer compiler errors for ambiguous method overloads when you are sure which method is better.

Enabling more efficient safe code

You should be able to write C# code safely that performs as well as unsafe code. Safe code avoids classes of errors, such as buffer overruns, stray pointers, and other memory access errors. These new features expand the capabilities of verifiable safe code. Strive to write more of your code using safe constructs. These features make that easier.

The following new features support the theme of better performance for safe code:

- You can access fixed fields without pinning.
- You can reassign `ref` local variables.
- You can use initializers on `stackalloc` arrays.
- You can use `fixed` statements with any type that supports a pattern.
- You can use additional generic constraints.
- The `in` modifier on parameters, to specify that an argument is passed by reference but not modified by the called method. Adding the `in` modifier to an argument is a [source compatible change](#).
- The `ref readonly` modifier on method returns, to indicate that a method returns its value by reference but doesn't allow writes to that object. Adding the `ref readonly` modifier is a [source compatible change](#), if the return is assigned to a value. Adding the `readonly` modifier to an existing `ref` return statement is an

incompatible change. It requires callers to update the declaration of `ref` local variables to include the `readonly` modifier.

- The `readonly struct` declaration, to indicate that a struct is immutable and should be passed as an `in` parameter to its member methods. Adding the `readonly` modifier to an existing struct declaration is a **binary compatible change**.
- The `ref struct` declaration, to indicate that a struct type accesses managed memory directly and must always be stack allocated. Adding the `ref` modifier to an existing `struct` declaration is an **incompatible change**. A `ref struct` cannot be a member of a class or used in other locations where it may be allocated on the heap.

You can read more about all these changes in [Write safe efficient code](#).

Ref locals and returns

This feature enables algorithms that use and return references to variables defined elsewhere. One example is working with large matrices, and finding a single location with certain characteristics. The following method returns a **reference** to that storage in the matrix:

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

You can declare the return value as a `ref` and modify that value in the matrix, as shown in the following code:

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

The C# language has several rules that protect you from misusing the `ref` locals and returns:

- You must add the `ref` keyword to the method signature and to all `return` statements in a method.
 - That makes it clear the method returns by reference throughout the method.
- A `ref return` may be assigned to a value variable, or a `ref` variable.
 - The caller controls whether the return value is copied or not. Omitting the `ref` modifier when assigning the return value indicates that the caller wants a copy of the value, not a reference to the storage.
- You can't assign a standard method return value to a `ref` local variable.
 - That disallows statements like `ref int i = sequence.Count();`
- You can't return a `ref` to a variable whose lifetime doesn't extend beyond the execution of the method.
 - That means you can't return a reference to a local variable or a variable with a similar scope.
- `ref` locals and returns can't be used with `async` methods.
 - The compiler can't know if the referenced variable has been set to its final value when the `async` method returns.

The addition of `ref` locals and `ref` returns enables algorithms that are more efficient by avoiding copying values, or performing dereferencing operations multiple times.

Adding `ref` to the return value is a [source compatible change](#). Existing code compiles, but the `ref` return value

is copied when assigned. Callers must update the storage for the return value to a `ref` local variable to store the return as a reference.

Now, `ref` locals may be reassigned to refer to different instances after being initialized. The following code now compiles:

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

For more information, see the article on [ref returns and ref locals](#), and the article on [foreach](#).

For more information, see the [ref keyword](#) article.

Conditional `ref` expressions

Finally, the conditional expression may produce a `ref` result instead of a value result. For example, you would write the following to retrieve a reference to the first element in one of two arrays:

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

The variable `r` is a reference to the first value in either `arr` or `otherArr`.

For more information, see [conditional operator \(?\)](#) in the language reference.

`in` parameter modifier

The `in` keyword complements the existing `ref` and `out` keywords to pass arguments by reference. The `in` keyword specifies passing the argument by reference, but the called method doesn't modify the value.

You may declare overloads that pass by value or by readonly reference, as shown in the following code:

```
static void M(S arg);
static void M(in S arg);
```

The by value (first in the preceding example) overload is better than the by readonly reference version. To call the version with the readonly reference argument, you must include the `in` modifier when calling the method.

For more information, see the article on the [in parameter modifier](#).

More types support the `fixed` statement

The `fixed` statement supported a limited set of types. Starting with C# 7.3, any type that contains a `GetPinnableReference()` method that returns a `ref T` or `ref readonly T` may be `fixed`. Adding this feature means that `fixed` can be used with `System.Span<T>` and related types.

For more information, see the [fixed statement](#) article in the language reference.

Indexing `fixed` fields does not require pinning

Consider this struct:

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

In earlier versions of C#, you needed to pin a variable to access one of the integers that are part of `myFixedField`. Now, the following code compiles without pinning the variable `p` inside a separate `fixed` statement:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

The variable `p` accesses one element in `myFixedField`. You don't need to declare a separate `int*` variable. You still need an `unsafe` context. In earlier versions of C#, you need to declare a second fixed pointer:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

For more information, see the article on the [fixed statement](#).

`stackalloc` arrays support initializers

You've been able to specify the values for elements in an array when you initialize it:

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

Now, that same syntax can be applied to arrays that are declared with `stackalloc`:

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

For more information, see the [stackalloc operator](#) article.

Enhanced generic constraints

You can now specify the type [System.Enum](#) or [System.Delegate](#) as base class constraints for a type parameter.

You can also use the new `unmanaged` constraint, to specify that a type parameter must be a non-nullable [unmanaged type](#).

For more information, see the articles on [where generic constraints](#) and [constraints on type parameters](#).

Adding these constraints to existing types is an [incompatible change](#). Closed generic types may no longer meet these new constraints.

Generalized async return types

Returning a `Task` object from async methods can introduce performance bottlenecks in certain paths. `Task` is a reference type, so using it means allocating an object. In cases where a method declared with the `async` modifier returns a cached result, or completes synchronously, the extra allocations can become a significant time

cost in performance critical sections of code. It can become costly if those allocations occur in tight loops.

The new language feature means that async method return types aren't limited to `Task`, `Task<T>`, and `void`. The returned type must still satisfy the async pattern, meaning a `GetAwaiter` method must be accessible. As one concrete example, the `ValueTask` type has been added to .NET to make use of this new language feature:

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
    return 5;
}
```

NOTE

You need to add the NuGet package [System.Threading.Tasks.Extensions](#) > in order to use the `ValueTask<TResult>` type.

This enhancement is most useful for library authors to avoid allocating a `Task` in performance critical code.

New compiler options

New compiler options support new build and DevOps scenarios for C# programs.

Reference assembly generation

There are two new compiler options that generate *reference-only assemblies*: [ProduceReferenceAssembly](#) and [ProduceOnlyReferenceAssembly](#). The linked articles explain these options and reference assemblies in more detail.

Public or Open Source signing

The `PublicSign` compiler option instructs the compiler to sign the assembly using a public key. The assembly is marked as signed, but the signature is taken from the public key. This option enables you to build signed assemblies from open-source projects using a public key.

For more information, see the [PublicSign compiler option](#) article.

pathmap

The `PathMap` compiler option instructs the compiler to replace source paths from the build environment with mapped source paths. The `PathMap` option controls the source path written by the compiler to PDB files or for the `CallerFilePathAttribute`.

For more information, see the [PathMap compiler option](#) article.

Learn about any breaking changes in the C# compiler

3/10/2021 • 2 minutes to read • [Edit Online](#)

The [Roslyn](#) team maintains a list of breaking changes in the C# and Visual Basic compilers. You can find information on those changes at these links on their GitHub repository:

- [Breaking changes in Roslyn after .NET 5](#)
- [Breaking changes in VS2019 version 16.8 introduced with .NET 5.0 and C# 9.0](#)
- [Breaking changes in VS2019 Update 1 and beyond compared to VS2019](#)
- [Breaking changes since VS2017 \(C# 7\)](#)
- [Breaking changes in Roslyn 3.0 \(VS2019\) from Roslyn 2.* \(VS2017\)](#)
- [Breaking changes in Roslyn 2.0 \(VS2017\) from Roslyn 1.* \(VS2015\) and native C# compiler \(VS2013 and previous\).](#)
- [Breaking changes in Roslyn 1.0 \(VS2015\) from the native C# compiler \(VS2013 and previous\).](#)
- [Unicode version change in C# 6](#)

The history of C#

3/23/2021 • 11 minutes to read • [Edit Online](#)

This article provides a history of each major release of the C# language. The C# team is continuing to innovate and add new features. Detailed language feature status, including features considered for upcoming releases can be found [on the dotnet/roslyn repository](#) on GitHub.

IMPORTANT

The C# language relies on types and methods in what the C# specification defines as a *standard library* for some of the features. The .NET platform delivers those types and methods in a number of packages. One example is exception processing. Every `throw` statement or expression is checked to ensure the object being thrown is derived from `Exception`. Similarly, every `catch` is checked to ensure that the type being caught is derived from `Exception`. Each version may add new requirements. To use the latest language features in older environments, you may need to install specific libraries. These dependencies are documented in the page for each specific version. You can learn more about the [relationships between language and library](#) for background on this dependency.

The C# build tools consider the latest major language release the default language version. There may be point releases between major releases, detailed in other articles in this section. To use the latest features in a point release, you need to [configure the compiler language version](#) and select the version. There have been three-point releases since C# 7.0:

- C# 7.3:
 - C# 7.3 is available starting with [Visual Studio 2017 version 15.7](#) and [.NET Core 2.1 SDK](#).
- C# 7.2:
 - C# 7.2 is available starting with [Visual Studio 2017 version 15.5](#) and [.NET Core 2.0 SDK](#).
- C# 7.1:
 - C# 7.1 is available starting with [Visual Studio 2017 version 15.3](#) and [.NET Core 2.0 SDK](#).

C# version 1.0

When you go back and look, C# version 1.0, released with Visual Studio .NET 2002, looked a lot like Java. As [part of its stated design goals for ECMA](#), it sought to be a "simple, modern, general-purpose object-oriented language." At the time, looking like Java meant it achieved those early design goals.

But if you look back on C# 1.0 now, you'd find yourself a little dizzy. It lacked the built-in async capabilities and some of the slick functionality around generics you take for granted. As a matter of fact, it lacked generics altogether. And [LINQ](#)? Not available yet. Those additions would take some years to come out.

C# version 1.0 looked stripped of features, compared to today. You'd find yourself writing some verbose code. But yet, you have to start somewhere. C# version 1.0 was a viable alternative to Java on the Windows platform.

The major features of C# 1.0 included:

- [Classes](#)
- [Structs](#)
- [Interfaces](#)
- [Events](#)
- [Properties](#)
- [Delegates](#)

- Operators and expressions
- Statements
- Attributes

C# version 1.2

C# version 1.2 shipped with Visual Studio .NET 2003. It contained a few small enhancements to the language. Most notable is that starting with this version, the code generated in a `foreach` loop called `Dispose` on an `IEnumerator` when that `IEnumerator` implemented `IDisposable`.

C# version 2.0

Now things start to get interesting. Let's take a look at some major features of C# 2.0, released in 2005, along with Visual Studio 2005:

- Generics
- Partial types
- Anonymous methods
- Nullable value types
- Iterators
- Covariance and contravariance

Other C# 2.0 features added capabilities to existing features:

- Getter/setter separate accessibility
- Method group conversions (delegates)
- Static classes
- Delegate inference

While C# may have started as a generic Object-Oriented (OO) language, C# version 2.0 changed that in a hurry. Once they had their feet under them, they went after some serious developer pain points. And they went after them in a significant way.

With generics, types and methods can operate on an arbitrary type while still retaining type safety. For instance, having a `List<T>` lets you have `List<string>` or `List<int>` and perform type-safe operations on those strings or integers while you iterate through them. Using generics is better than creating a `ListInt` type which derives from `ArrayList` or casting from `object` for every operation.

C# version 2.0 brought iterators. To put it succinctly, iterators let you examine all the items in a `List` (or other Enumerable types) with a `foreach` loop. Having iterators as a first-class part of the language dramatically enhanced readability of the language and people's ability to reason about the code.

And yet, C# continued to play a bit of catch-up with Java. Java had already released versions that included generics and iterators. But that would soon change as the languages continued to evolve apart.

C# version 3.0

C# version 3.0 came in late 2007, along with Visual Studio 2008, though the full boat of language features would actually come with .NET Framework version 3.5. This version marked a major change in the growth of C#. It established C# as a truly formidable programming language. Let's take a look at some major features in this version:

- Auto-implemented properties
- Anonymous types

- [Query expressions](#)
- [Lambda expressions](#)
- [Expression trees](#)
- [Extension methods](#)
- [Implicitly typed local variables](#)
- [Partial methods](#)
- [Object and collection initializers](#)

In retrospect, many of these features seem both inevitable and inseparable. They all fit together strategically. It's generally thought that C# version's killer feature was the query expression, also known as Language-Integrated Query (LINQ).

A more nuanced view examines expression trees, lambda expressions, and anonymous types as the foundation upon which LINQ is constructed. But, in either case, C# 3.0 presented a revolutionary concept. C# 3.0 had begun to lay the groundwork for turning C# into a hybrid Object-Oriented / Functional language.

Specifically, you could now write SQL-style, declarative queries to perform operations on collections, among other things. Instead of writing a `for` loop to compute the average of a list of integers, you could now do that as simply as `list.Average()`. The combination of query expressions and extension methods made it look as though that list of integers had gotten a whole lot smarter.

It took time for people to really grasp and integrate the concept, but they gradually did. And now, years later, code is much more concise, simple, and functional.

C# version 4.0

C# version 4.0, released with Visual Studio 2010, would have had a difficult time living up to the groundbreaking status of version 3.0. With version 3.0, C# had moved the language firmly out from the shadow of Java and into prominence. The language was quickly becoming elegant.

The next version did introduce some interesting new features:

- [Dynamic binding](#)
- [Named/optional arguments](#)
- [Generic covariant and contravariant](#)
- [Embedded interop types](#)

Embedded interop types alleviated a deployment pain. Generic covariance and contravariance give you more power to use generics, but they're a bit academic and probably most appreciated by framework and library authors. Named and optional parameters let you eliminate many method overloads and provide convenience. But none of those features are exactly paradigm altering.

The major feature was the introduction of the `dynamic` keyword. The `dynamic` keyword introduced into C# version 4.0 the ability to override the compiler on compile-time typing. By using the `dynamic` keyword, you can create constructs similar to dynamically typed languages like JavaScript. You can create a `dynamic x = "a string"` and then add six to it, leaving it up to the runtime to sort out what should happen next.

Dynamic binding gives you the potential for errors but also great power within the language.

C# version 5.0

C# version 5.0, released with Visual Studio 2012, was a focused version of the language. Nearly all of the effort for that version went into another groundbreaking language concept: the `async` and `await` model for asynchronous programming. Here is the major features list:

- [Asynchronous members](#)
- [Caller info attributes](#)

See Also

- [Code Project: Caller Info Attributes in C# 5.0](#)

The caller info attribute lets you easily retrieve information about the context in which you're running without resorting to a ton of boilerplate reflection code. It has many uses in diagnostics and logging tasks.

But `async` and `await` are the real stars of this release. When these features came out in 2012, C# changed the game again by baking asynchrony into the language as a first-class participant. If you've ever dealt with long running operations and the implementation of webs of callbacks, you probably loved this language feature.

C# version 6.0

With versions 3.0 and 5.0, C# had added major new features in an object-oriented language. With version 6.0, released with Visual Studio 2015, it would go away from doing a dominant killer feature and instead release many smaller features that made C# programming more productive. Here are some of them:

- [Static imports](#)
- [Exception filters](#)
- [Auto-property initializers](#)
- [Expression bodied members](#)
- [Null propagator](#)
- [String interpolation](#)
- [nameof operator](#)

Other new features include:

- Index initializers
- Await in catch/finally blocks
- Default values for getter-only properties

Each of these features is interesting in its own right. But if you look at them altogether, you see an interesting pattern. In this version, C# eliminated language boilerplate to make code more terse and readable. So for fans of clean, simple code, this language version was a huge win.

They did one other thing along with this version, though it's not a traditional language feature in itself. They released [Roslyn the compiler as a service](#). The C# compiler is now written in C#, and you can use the compiler as part of your programming efforts.

C# version 7.0

C# version 7.0 was released with Visual Studio 2017. This version has some evolutionary and cool stuff in the vein of C# 6.0, but without the compiler as a service. Here are some of the new features:

- [Out variables](#)
- [Tuples and deconstruction](#)
- [Pattern matching](#)
- [Local functions](#)
- [Expanded expression bodied members](#)
- [Ref locals and returns](#)

Other features included:

- [Discards](#)
- [Binary Literals and Digit Separators](#)
- [Throw expressions](#)

All of these features offer cool new capabilities for developers and the opportunity to write even cleaner code than ever. A highlight is condensing the declaration of variables to use with the `out` keyword and by allowing multiple return values via tuple.

But C# is being put to ever broader use. .NET Core now targets any operating system and has its eyes firmly on the cloud and on portability. These new capabilities certainly occupy the language designers' thoughts and time, in addition to coming up with new features.

C# version 7.1

C# started releasing *point releases* with C# 7.1. This version added the [language version selection](#) configuration element, three new language features, and new compiler behavior.

The new language features in this release are:

- `async` `Main` method
 - The entry point for an application can have the `async` modifier.
- `default` literal expressions
 - You can use default literal expressions in default value expressions when the target type can be inferred.
- Inferred tuple element names
 - The names of tuple elements can be inferred from tuple initialization in many cases.
- Pattern matching on generic type parameters
 - You can use pattern match expressions on variables whose type is a generic type parameter.

Finally, the compiler has two options `-refout` and `-refonly` that control [reference assembly generation](#).

C# version 7.2

C# 7.2 added several small language features:

- [Techniques for writing safe efficient code](#)
 - A combination of syntax improvements that enable working with value types using reference semantics.
- [Non-trailing named arguments](#)
 - Named arguments can be followed by positional arguments.
- [Leading underscores in numeric literals](#)
 - Numeric literals can now have leading underscores before any printed digits.
- `private protected` access modifier
 - The `private protected` access modifier enables access for derived classes in the same assembly.
- [Conditional ref expressions](#)
 - The result of a conditional expression (`? :`) can now be a reference.

C# version 7.3

There are two main themes to the C# 7.3 release. One theme provides features that enable safe code to be as performant as unsafe code. The second theme provides incremental improvements to existing features. In addition, new compiler options were added in this release.

The following new features support the theme of better performance for safe code:

- You can access fixed fields without pinning.
- You can reassign `ref` local variables.
- You can use initializers on `stackalloc` arrays.
- You can use `fixed` statements with any type that supports a pattern.
- You can use additional generic constraints.

The following enhancements were made to existing features:

- You can test `==` and `!=` with tuple types.
- You can use expression variables in more locations.
- You may attach attributes to the backing field of auto-implemented properties.
- Method resolution when arguments differ by `in` has been improved.
- Overload resolution now has fewer ambiguous cases.

The new compiler options are:

- `-publicsign` to enable Open Source Software (OSS) signing of assemblies.
- `-pathmap` to provide a mapping for source directories.

C# version 8.0

C# 8.0 is the first major C# release that specifically targets .NET Core. Some features rely on new CLR capabilities, others on library types added only in .NET Core. C# 8.0 adds the following features and enhancements to the C# language:

- Readonly members
- Default interface methods
- Pattern matching enhancements:
 - Switch expressions
 - Property patterns
 - Tuple patterns
 - Positional patterns
- Using declarations
- Static local functions
- Disposable ref structs
- Nullable reference types
- Asynchronous streams
- Indices and ranges
- Null-coalescing assignment
- Unmanaged constructed types
- Stackalloc in nested expressions
- Enhancement of interpolated verbatim strings

Default interface members require enhancements in the CLR. Those features were added in the CLR for .NET Core 3.0. Ranges and indexes, and asynchronous streams require new types in the .NET Core 3.0 libraries.

Nullable reference types, while implemented in the compiler, is much more useful when libraries are annotated to provide semantic information regarding the null state of arguments and return values. Those annotations are being added in the .NET Core libraries.

Relationships between language features and library types

11/2/2020 • 2 minutes to read • [Edit Online](#)

The C# language definition requires a standard library to have certain types and certain accessible members on those types. The compiler generates code that uses these required types and members for many different language features. When necessary, there are NuGet packages that contain types needed for newer versions of the language when writing code for environments where those types or members have not been deployed yet.

This dependency on standard library functionality has been part of the C# language since its first version. In that version, examples included:

- [Exception](#) - used for all compiler generated exceptions.
- [String](#) - the C# `string` type is a synonym for [String](#).
- [Int32](#) - synonym of `int`.

That first version was simple: the compiler and the standard library shipped together, and there was only one version of each.

Subsequent versions of C# have occasionally added new types or members to the dependencies. Examples include: [INotifyCompletion](#), [CallerFilePathAttribute](#) and [CallerMemberNameAttribute](#). C# 7.0 continues this by adding a dependency on [ValueTuple](#) to implement the [tuples](#) language feature.

The language design team works to minimize the surface area of the types and members required in a compliant standard library. That goal is balanced against a clean design where new library features are incorporated seamlessly into the language. There will be new features in future versions of C# that require new types and members in a standard library. It's important to understand how to manage those dependencies in your work.

Managing your dependencies

C# compiler tools are now decoupled from the release cycle of the .NET libraries on supported platforms. In fact, different .NET libraries have different release cycles: the .NET Framework on Windows is released as a Windows Update, .NET Core ships on a separate schedule, and the Xamarin versions of library updates ship with the Xamarin tools for each target platform.

The majority of time, you won't notice these changes. However, when you are working with a newer version of the language that requires features not yet in the .NET libraries on that platform, you'll reference the NuGet packages to provide those new types. As the platforms your app supports are updated with new framework installations, you can remove the extra reference.

This separation means you can use new language features even when you are targeting machines that may not have the corresponding framework.

Version and update considerations for C# developers

11/2/2020 • 2 minutes to read • [Edit Online](#)

Compatibility is a very important goal as new features are added to the C# language. In almost all cases, existing code can be recompiled with a new compiler version without any issue.

More care may be required when you adopt new language features in a library. You may be creating a new library with features found in the latest version and need to ensure apps built using previous versions of the compiler can use it. Or you may be upgrading an existing library and many of your users may not have upgraded versions yet. As you make decisions on adopting new features, you'll need to consider two variations of compatibility: source compatible and binary compatible.

Binary compatible changes

Changes to your library are **binary compatible** when your updated library can be used without rebuilding applications and libraries that use it. Dependent assemblies are not required to be rebuilt, nor are any source code changes required. Binary compatible changes are also source compatible changes.

Source compatible changes

Changes to your library are **source compatible** when applications and libraries that use your library do not require source code changes, but the source must be recompiled against the new version to work correctly.

Incompatible changes

If a change is neither **source compatible** nor **binary compatible**, source code changes along with recompilation are required in dependent libraries and applications.

Evaluate your library

These compatibility concepts affect the public and protected declarations for your library, not its internal implementation. Adopting any new features internally are always **binary compatible**.

Binary compatible changes provide new syntax that generates the same compiled code for public declarations as the older syntax. For example, changing a method to an expression-bodied member is a **binary compatible** change:

Original code:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

New code:

```
public double CalculateSquare(double value) => value * value;
```

Source compatible changes introduce syntax that changes the compiled code for a public member, but in a

way that is compatible with existing call sites. For example, changing a method signature from a by value parameter to an `in` by reference parameter is source compatible, but not binary compatible:

Original code:

```
public double CalculateSquare(double value) => value * value;
```

New code:

```
public double CalculateSquare(in double value) => value * value;
```

The [What's new](#) articles note if introducing a feature that affects public declarations is source compatible or binary compatible.

Create record types

4/3/2021 • 11 minutes to read • [Edit Online](#)

C# 9 introduces *records*, a new reference type that you can create instead of classes or structs. Records are distinct from classes in that record types use *value-based equality*. Two variables of a record type are equal if the record type definitions are identical, and if for every field, the values in both records are equal. Two variables of a class type are equal if the objects referred to are the same class type and the variables refer to the same object. Value-based equality implies other capabilities you'll probably want in record types. The compiler generates many of those members when you declare a `record` instead of a `class`.

In this tutorial, you'll learn how to:

- Decide if you should declare a `class` or a `record`.
- Declare record types and positional record types.
- Substitute your methods for compiler generated methods in records.

Prerequisites

You'll need to set up your machine to run .NET 5 or later, including the C# 9.0 or later compiler. The C# 9.0 compiler is available starting with [Visual Studio 2019 version 16.8](#) or the [.NET 5.0 SDK](#).

Characteristics of records

You define a *record* by declaring a type with the `record` keyword, instead of the `class` or `struct` keyword. A record is a reference type and follows value-based equality semantics. To enforce value semantics, the compiler generates several methods for your record type:

- An override of [Object.Equals\(Object\)](#).
- A virtual `Equals` method whose parameter is the record type.
- An override of [Object.GetHashCode\(\)](#).
- Methods for `operator ==` and `operator !=`.
- Record types implement [System.IEquatable<T>](#).

In addition, records provide an override of [Object.ToString\(\)](#). The compiler synthesizes methods for displaying records using [Object.ToString\(\)](#). You'll explore those members as you write the code for this tutorial. Records support `with` expressions to enable non-destructive mutation of records.

You can also declare *positional records* using a more concise syntax. The compiler synthesizes more methods for you when you declare positional records:

- A primary constructor whose parameters match the positional parameters on the record declaration.
- Public init-only properties for each parameter of a primary constructor.
- A `Deconstruct` method to extract properties from the record.

Build temperature data

Data and statistics are among the scenarios where you'll want to use records. For this tutorial, you'll build an application that computes *degree days* for different uses. *Degree days* are a measure of heat (or lack of heat) over a period of days, weeks, or months. Degree days track and predict energy usage. More hotter days means more air conditioning, and more colder days means more furnace usage. Degree days help manage plant

populations and correlate to plant growth as the seasons change. Degree days help track animal migrations for species that travel to match climate.

The formula is based on the mean temperature on a given day and a baseline temperature. To compute degree days over time, you'll need the high and low temperature each day for a period of time. Let's start by creating a new application. Make a new console application. Create a new record type in a new file named "DailyTemperature.cs":

```
public record DailyTemperature(double HighTemp, double LowTemp);
```

The preceding code defines a *positional record*. You've created a reference type that contains two properties: `HighTemp`, and `LowTemp`. Those properties are *init only properties*, meaning they can be set in the constructor or using a property initializer. The `DailyTemperature` type also has a *primary constructor* that has two parameters that match the two properties. You use the primary constructor to initialize a `DailyTemperature` record:

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
};
```

You can add your own properties or methods to records, including positional records. You'll need to compute the mean temperature for each day. You can add that property to the `DailyTemperature` record:

```
public record DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Let's make sure you can use this data. Add the following code to your `Main` method:

```
foreach (var item in data)
    Console.WriteLine(item);
```

Run your application, and you'll see output that looks similar to the following display (several rows removed for space):

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
```

```
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }
```

```
DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
```

```
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

The preceding code shows the output from the override of `ToString` synthesized by the compiler. If you prefer different text, you can write your own version of `ToString` that prevents the compiler from synthesizing a version for you.

Compute degree days

To compute degree days, you take the difference from a baseline temperature and the mean temperature on a given day. To measure heat over time, you discard any days where the mean temperature is below the baseline. To measure cold over time, you discard any days where the mean temperature is above the baseline. For example, the U.S. uses 65F as the base for both heating and cooling degree days. That's the temperature where no heating or cooling is needed. If a day has a mean temperature of 70F, that day is 5 cooling degree days and 0 heating degree days. Conversely, if the mean temperature is 55F, that day is 10 heating degree days and 0 cooling degree days.

You can express these formulas as a small hierarchy of record types: an abstract degree day type and two concrete types for heating degree days and cooling degree days. These types can also be positional records. They take a baseline temperature and a sequence of daily temperature records as arguments to the primary constructor:

```
public abstract record DegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords);

public record HeatingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean < BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean > BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

The abstract `DegreeDays` record is the shared base class for both the `HeatingDegreeDays` and `CoolingDegreeDays` records. The primary constructor declarations on the derived records show how to manage base record initialization. Your derived record declares parameters for all the parameters in the base record primary constructor. The base record declares and initializes those properties. The derived record doesn't hide them, but only creates and initializes properties for parameters that aren't declared in its base record. In this example, the derived records don't add new primary constructor parameters. Test your code by adding the following code to your `Main` method:

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

You'll get output like the following display:

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Define compiler-synthesized methods

Your code calculates the correct number of heating and cooling degree days over that period of time. But this example shows why you may want to replace some of the synthesized methods for records. You can declare your own version of any of the compiler-synthesized methods in a record type except the clone method. The clone method has a compiler generated name and you cannot provide a different implementation. These synthesized methods include a copy constructor, the members of the [System.IEquatable<T>](#) interface, equality and inequality tests, and [GetHashCode\(\)](#). For this purpose, you'll synthesize `PrintMembers`. You could also declare your own `ToString`, but `PrintMembers` provides a better option for inheritance scenarios. To provide your own version of a synthesized method, the signature must match the synthesized method.

The `TempRecords` element in the console output isn't useful. It displays the type, but nothing else. You can change this behavior by providing your own implementation of the synthesized `PrintMembers` method. The signature depends on modifiers applied to the `record` declaration:

- If a record type is `sealed`, the signature is `private bool PrintMembers(StringBuilder builder);`
- If a record type isn't `sealed` and derives from `object` (that is, it doesn't declare a base record), the signature is `protected virtual bool PrintMembers(StringBuilder builder);`
- If a record type isn't `sealed` and derives from another record, the signature is
`protected override bool PrintMembers(StringBuilder builder);`

These rules are easiest to comprehend through understanding the purpose of `PrintMembers`. `PrintMembers` adds information about each property in a record type to a string. The contract requires base records to add their members to the display and assumes derived members will add their members. Each record type synthesizes a `ToString` override that looks similar to the following example for `HeatingDegreeDays`:

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

You declare a `PrintMembers` method in the `DegreeDays` record that doesn't print the type of the collection:

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

The signature declares a `virtual protected` method to match the compiler's version. Don't worry if you get the accessors wrong; the language enforces the correct signature. If you forget the correct modifiers for any synthesized method, the compiler issues warnings or errors that help you get the right signature.

Non-destructive mutation

The synthesized members in a positional record don't modify the state of the record. The goal is that you can more easily create immutable records. Look again at the preceding declarations for `HeatingDegreeDays` and `CoolingDegreeDays`. The members added perform computations on the values for the record, but don't mutate state. Positional records make it easier for you to create immutable reference types.

Creating immutable reference types means you'll want to use non-destructive mutation. You create new record instances that are similar to existing record instances using `with` expressions. These expressions are a copy construction with additional assignments that modify the copy. The result is a new record instance where each property has been copied from the existing record and optionally modified. The original record is unchanged.

Let's add a couple features to your program that demonstrate `with` expressions. First, let's create a new record to compute growing degree days using the same data. *Growing degree days* typically uses 41F as the baseline and measures temperatures above the baseline. To use the same data, you can create a new record that is similar to the `coolingDegreeDays`, but with a different base temperature:

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

You can compare the number of degrees computed to the numbers generated with a higher baseline temperature. Remember that records are *reference types* and these copies are shallow copies. The array for the data isn't copied, but both records refer to the same data. That fact is an advantage in one other scenario. For growing degree days, it's useful to keep track of the total for the previous 5 days. You can create new records with different source data using `with` expressions. The following code builds a collection of these accumulations, then displays the values:

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

You can also use `with` expressions to create copies of records. Don't specify any properties between the braces for the `with` expression. That means create a copy, and don't change any properties:

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Run the finished application to see the results.

Summary

This tutorial showed several aspects of records. Records provide concise syntax for reference types where the fundamental use is storing data. For object-oriented classes, the fundamental use is defining responsibilities. This tutorial focused on *positional records*, where you can use a concise syntax to declare the init-only

properties for a record. The compiler synthesizes several members of the record for copying and comparing records. You can add any other members you need for your record types. You can create immutable record types knowing that none of the compiler-generated members would mutate state. And `with` expressions make it easy to support non-destructive mutation.

Records add another way to define types. You use `class` definitions to create object-oriented hierarchies that focus on the responsibilities and behavior of objects. You create `struct` types for data structures that store data and are small enough to copy efficiently. You create `record` types when you want value-based equality and comparison, don't want to copy values, and want to use reference variables.

You can learn the complete description of records by reading the [C# language reference article for the record type](#)) and the [proposed record type specification](#).

Tutorial: Explore ideas using top-level statements to build code as you learn

3/23/2021 • 7 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to:

- Learn the rules governing your use of top-level statements.
- Use top-level statements to explore algorithms.
- Refactor explorations into reusable components.

Prerequisites

You'll need to set up your machine to run .NET 5, which includes the C# 9 compiler. The C# 9 compiler is available starting with [Visual Studio 2019 version 16.9 preview 1](#) or [.NET 5.0 SDK](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Start exploring

Top-level statements enable you to avoid the extra ceremony required by placing your program's entry point in a static method in a class. The typical starting point for a new console application looks like the following code:

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The preceding code is the result of running the `dotnet new console` command and creating a new console application. Those 11 lines contain only one line of executable code. You can simplify that program with the new top-level statements feature. That enables you to remove all but two of the lines in this program:

```
using System;

Console.WriteLine("Hello World!");
```

This feature simplifies what's needed to begin exploring new ideas. You can use top-level statements for scripting scenarios, or to explore. Once you've got the basics working, you can start refactoring the code and create methods, classes, or other assemblies for reusable components you've built. Top-level statements do enable quick experimentation and beginner tutorials. They also provide a smooth path from experimentation to full programs.

Top-level statements are executed in the order they appear in the file. Top-level statements can only be used in one source file in your application. The compiler generates an error if you use them in more than one file.

Build a magic .NET answer machine

For this tutorial, let's build a console application that answers a "yes" or "no" question with a random answer. You'll build out the functionality step by step. You can focus on your task rather than ceremony needed for the structure of a typical program. Then, once you're happy with the functionality, you can refactor the application as you see fit.

A good starting point is to write the question back to the console. You can start by writing the following code:

```
using System;

Console.WriteLine(args);
```

You don't declare an `args` variable. For the single source file that contains your top-level statements, the compiler recognizes `args` to mean the command-line arguments. The type of `args` is a `string[]`, as in all C# programs.

You can test your code by running the following `dotnet run` command:

```
dotnet run -- Should I use top level statements in all my programs?
```

The arguments after the `--` on the command line are passed to the program. You can see the type of the `args` variable, because that's what's printed to the console:

```
System.String[]
```

To write the question to the console, you'll need to enumerate the arguments and separate them with a space. Replace the `WriteLine` call with the following code:

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Now, when you run the program, it will correctly display the question as a string of arguments.

Respond with a random answer

After echoing the question, you can add the code to generate the random answer. Start by adding an array of possible answers:

```

string[] answers =
{
    "It is certain.",      "Reply hazy, try again.",      "Don't count on it.",
    "It is decidedly so.", "Ask again later.",           "My reply is no.",
    "Without a doubt.",   "Better not tell you now.",   "My sources say no.",
    "Yes - definitely.",  "Cannot predict now.",        "Outlook not so good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

```

This array has 12 answers that are affirmative, six that are non-committal, and six that are negative. Next, add the following code to generate and display a random answer from the array:

```

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);

```

You can run the application again to see the results. You should see something like the following output:

```

dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.

```

This code answers the questions, but let's add one more feature. You'd like your question app to simulate thinking about the answer. You can do that by adding a bit of ASCII animation, and pausing while working. Add the following code after the line that echoes the question:

```

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

```

You'll also need to add a `using` statement to the top of the source file:

```

using System.Threading.Tasks;

```

The `using` statements must be before any other statements in the file. Otherwise, it's a compiler error. You can run the program again and see the animation. That makes a better experience. Experiment with the length of the delay to match your taste.

The preceding code creates a set of spinning lines separated by a space. Adding the `await` keyword instructs

the compiler to generate the program entry point as a method that has the `async` modifier, and returns a `System.Threading.Tasks.Task`. This program does not return a value, so the program entry point returns a `Task`. If your program returns an integer value, you would add a return statement to the end of your top-level statements. That return statement would specify the integer value to return. If your top-level statements include an `await` expression, the return type becomes `System.Threading.Tasks.Task<TResult>`.

Refactoring for the future

Your program should look like the following code:

```
using System;
using System.Threading.Tasks;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
{
    "It is certain.",      "Reply hazy, try again.",      "Don't count on it.",
    "It is decidedly so.",  "Ask again later.",            "My reply is no.",
    "Without a doubt.",    "Better not tell you now.",    "My sources say no.",
    "Yes - definitely.",   "Cannot predict now.",         "Outlook not so good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

The preceding code is reasonable. It works. But it isn't reusable. Now that you have the application working, it's time to pull out reusable parts.

One candidate is the code that displays the waiting animation. That snippet can become a method:

You can start by creating a local function in your file. Replace the current animation with the following code:

```

await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}

```

The preceding code creates a local function inside your main method. That's still not reusable. So, extract that code into a class. Create a new file named *utilities.cs* and add the following code:

```

using System;
using System.Threading.Tasks;

namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}

```

A file that has top-level statements can also contain namespaces and types at the end of the file, after the top-level statements. But for this tutorial you put the animation method in a separate file to make it more readily reusable.

Finally, you can clean the animation code to remove some duplication:

```

foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.WriteLine(s);
    await Task.Delay(50);
    Console.WriteLine("\b\b\b");
}

```

Now you have a complete application, and you've refactored the reusable parts for later use. You can call the new utility method from your top-level statements, as shown below in the finished version of the main program:

```

using System;
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.WriteLine(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
{
    "It is certain.",      "Reply hazy, try again.",      "Don't count on it.",
    "It is decidedly so.", "Ask again later.",            "My reply is no.",
    "Without a doubt.",   "Better not tell you now.",    "My sources say no.",
    "Yes - definitely.",  "Cannot predict now.",         "Outlook not so good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);

```

This adds the call to `Utilities.ShowConsoleAnimation`, and adds an additional `using` statement.

Summary

Top-level statements make it easier to create simple programs for use to explore new algorithms. You can experiment with algorithms by trying different snippets of code. Once you've learned what works, you can refactor the code to be more maintainable.

Top-level statements simplify programs that are based on console applications. These include Azure functions, GitHub actions, and other small utilities. For more information, see [Top-level statements \(C# Programming Guide\)](#).

Use pattern matching to build your class behavior for better code

3/23/2021 • 10 minutes to read • [Edit Online](#)

The pattern matching features in C# provide syntax to express your algorithms. You can use these techniques to implement the behavior in your classes. You can combine object-oriented class design with a data-oriented implementation to provide concise code while modeling real-world objects.

In this tutorial, you'll learn how to:

- Express your object oriented classes using data patterns.
- Implement those patterns using C#'s pattern matching features.
- Leverage compiler diagnostics to validate your implementation.

Prerequisites

You'll need to set up your machine to run .NET 5, including the C# 9.0 compiler. The C# 9.0 compiler is available starting with [Visual Studio 2019 version 16.8 preview](#) or the [.NET 5.0 SDK preview](#).

Build a simulation of a canal lock

In this tutorial, you'll build a C# class that simulates a [canal lock](#). Briefly, a canal lock is a device that raises and lowers boats as they travel between two stretches of water at different levels. A lock has two gates and some mechanism to change the water level.

In its normal operation, a boat enters one of the gates while the water level in the lock matches the water level on the side the boat enters. Once in the lock, the water level is changed to match the water level where the boat will leave the lock. Once the water level matches that side, the gate on the exit side opens. Safety measures make sure an operator can't create a dangerous situation in the canal. The water level can be changed only when both gates are closed. At most one gate can be open. To open a gate, the water level in the lock must match the water level outside the gate being opened.

You can build a C# class to model this behavior. A `CanalLock` class would support commands to open or close either gate. It would have other commands to raise or lower the water. The class should also support properties to read the current state of both gates and the water level. Your methods implement the safety measures.

Define a class

You'll build a console application to test your `CanalLock` class. Create a new console project for .NET 5 using either Visual Studio or the .NET CLI. Then, add a new class and name it `CanalLock`. Next, design your public API, but leave the methods not implemented:

```

public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } = WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}

```

The preceding code initializes the object so both gates are closed, and the water level is low. Next, write the following test code in your `Main` method to guide you as you create a first implementation of the class:

```

// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");
Console.WriteLine(canalGate);

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

```

Next, add a first implementation of each method in the `CanalLock` class. The following code implements the methods of the class without concern to the safety rules. You'll add safety tests later:

```

// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}

```

The tests you've written so far pass. You've implemented the basics. Now, write a test for the first failure condition. At the end of the previous tests, both gates are closed, and the water level is set to low. Add a test to try opening the upper gate:

```

Console.WriteLine("=====");
Console.WriteLine("    Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");

```

This test fails because the gate opens. As a first implementation, you could fix it with the following code:

```

// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the water is low");
}

```

Your tests pass. But, as you add more tests, you'll add more and more `if` clauses and test different properties. Soon, these methods will get too complicated as you add more conditionals.

Implement the commands with patterns

A better way is to use *patterns* to determine if the object is in a valid state to execute a command. You can express if a command is allowed as a function of three variables: the state of the gate, the level of the water, and the new setting:

NEW SETTING	GATE STATE	WATER LEVEL	RESULT
Closed	Closed	High	Closed
Closed	Closed	Low	Closed
Closed	Open	High	Open
Closed	Open	Low	Closed
Open	Closed	High	Open
Open	Closed	Low	Closed (Error)
Open	Open	High	Open
Open	Open	Low	Closed (Error)

The fourth and last rows in the table have strike through text because they're invalid. The code you're adding now should make sure the high water gate is never opened when the water is low. Those states can be coded as a single switch expression (remember that `false` indicates "Closed"):

```

HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the
water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};

```

Try this version. Your tests pass, validating the code. The full table shows the possible combinations of inputs and results. That means you and other developers can quickly look at the table and see that you've covered all the possible inputs. Even easier, the compiler can help as well. After you add the previous code, you can see that the compiler generates a warning: *CS8524* indicates the switch expression doesn't cover all possible inputs. The reason for that warning is that one of the inputs is an `enum` type. The compiler interprets "all possible inputs" as all inputs from the underlying type, typically an `int`. This `switch` expression only checks the values declared in the `enum`. To remove the warning, you can add a catch-all discard pattern for the last arm of the expression. This condition throws an exception, because it indicates invalid input:

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

The preceding switch arm must be last in your `switch` expression because it matches all inputs. Experiment by moving it earlier in the order. That causes a compiler error *CS8510* for unreachable code in a pattern. The natural structure of switch expressions enables the compiler to generate errors and warnings for possible mistakes. The compiler "safety net" makes it easier for you to create correct code in fewer iterations, and the freedom to combine switch arms with wildcards. The compiler will issue errors if your combination results in unreachable arms you didn't expect, and warnings if you remove an arm that's needed.

The first change is to combine all the arms where the command is to close the gate; that's always allowed. Add the following code as the first arm in your switch expression:

```
(false, _, _) => false,
```

After you add the previous switch arm, you'll get four compiler errors, one on each of the arms where the command is `false`. Those arms are already covered by the newly added arm. You can safely remove those four lines. You intended this new switch arm to replace those conditions.

Next, you can simplify the four arms where the command is to open the gate. In both cases where the water level is high, the gate can be opened. (In one, it's already open.) One case where the water level is low throws an exception, and the other shouldn't happen. It should be safe to throw the same exception if the water lock is already in an invalid state. You can make the following simplifications for those arms:

```

(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the water
is low"),
_ => throw new InvalidOperationException("Invalid internal state"),

```

Run your tests again, and they pass. Here's the final version of the `SetHighGate` method:

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _)           => false,
        (true, _, WaterLevel.High) => true,
        (true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when
the water is low"),
        _                      => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Implement patterns yourself

Now that you've seen the technique, fill in the `SetLowGate` and `SetWaterLevel` methods yourself. Start by adding the following code to test invalid operations on those methods:

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Run your application again. You can see the new tests fail, and the canal lock gets into an invalid state. Try to implement the remaining methods yourself. The method to set the lower gate should be similar to the method to set the upper gate. The method that changes the water level has different checks, but should follow a similar

structure. You may find it helpful to use the same process for the method that sets the water level. Start with all four inputs: The state of both gates, the current state of the water level, and the requested new water level. The switch expression should start with:

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};
```

You'll have 16 total switch arms to fill in. Then, test and simplify.

Did you make methods something like this?

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new InvalidOperationException("Cannot open high gate when
the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new InvalidOperationException("Cannot lower
water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new InvalidOperationException("Cannot raise
water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Your tests should pass, and the canal lock should operate safely.

Summary

In this tutorial, you learned to use pattern matching to check the internal state of an object before applying any changes to that state. You can check combinations of properties. Once you've built tables for any of those transitions, you test your code, then simplify for readability and maintainability. These initial refactorings may suggest further refactorings that validate internal state or manage other API changes. This tutorial combined classes and objects with a more data-oriented, pattern-based approach to implement those classes.

Tutorial: Update interfaces with default interface methods in C# 8.0

3/23/2021 • 6 minutes to read • [Edit Online](#)

Beginning with C# 8.0 on .NET Core 3.0, you can define an implementation when you declare a member of an interface. The most common scenario is to safely add members to an interface already released and used by innumerable clients.

In this tutorial, you'll learn how to:

- Extend interfaces safely by adding methods with implementations.
- Create parameterized implementations to provide greater flexibility.
- Enable implementers to provide a more specific implementation in the form of an override.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8.0 compiler is available starting with [Visual Studio 2019 version 16.3](#) or the [.NET Core 3.0 SDK](#).

Scenario overview

This tutorial starts with version 1 of a customer relationship library. You can get the starter application on our [samples repo on GitHub](#). The company that built this library intended customers with existing applications to adopt their library. They provided minimal interface definitions for users of their library to implement. Here's the interface definition for a customer:

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

They defined a second interface that represents an order:

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

From those interfaces, the team could build a library for their users to create a better experience for their customers. Their goal was to create a deeper relationship with existing customers and improve their relationships with new customers.

Now, it's time to upgrade the library for the next release. One of the requested features enables a loyalty discount for customers that have lots of orders. This new loyalty discount gets applied whenever a customer makes an order. The specific discount is a property of each individual customer. Each implementation of

`ICustomer` can set different rules for the loyalty discount.

The most natural way to add this functionality is to enhance the `ICustomer` interface with a method to apply any loyalty discount. This design suggestion caused concern among experienced developers: "Interfaces are immutable once they've been released! This is a breaking change!" C# 8.0 adds *default interface implementations* for upgrading interfaces. The library authors can add new members to the interface and provide a default implementation for those members.

Default interface implementations enable developers to upgrade an interface while still enabling any implementors to override that implementation. Users of the library can accept the default implementation as a non-breaking change. If their business rules are different, they can override.

Upgrade with default interface methods

The team agreed on the most likely default implementation: a loyalty discount for customers.

The upgrade should provide the functionality to set two properties: the number of orders needed to be eligible for the discount, and the percentage of the discount. This makes it a perfect scenario for default interface methods. You can add a method to the `ICustomer` interface, and provide the most likely implementation. All existing, and any new implementations can use the default implementation, or provide their own.

First, add the new method to the interface, including the body of the method:

```
// Version 1:  
public decimal ComputeLoyaltyDiscount()  
{  
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);  
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))  
    {  
        return 0.10m;  
    }  
    return 0;  
}
```

The library author wrote a first test to check the implementation:

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))  
{  
    Reminders =  
    {  
        { new DateTime(2010, 08, 12), "child's birthday" },  
        { new DateTime(1012, 11, 15), "anniversary" }  
    }  
};  
  
SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);  
c.AddOrder(o);  
  
o = new SampleOrder(new DateTime(2103, 7, 4), 25m);  
c.AddOrder(o);  
  
// Check the discount:  
ICustomer theCustomer = c;  
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Notice the following portion of the test:

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

That cast from `SampleCustomer` to `ICustomer` is necessary. The `SampleCustomer` class doesn't need to provide an implementation for `ComputeLoyaltyDiscount`; that's provided by the `ICustomer` interface. However, the `SampleCustomer` class doesn't inherit members from its interfaces. That rule hasn't changed. In order to call any method declared and implemented in the interface, the variable must be the type of the interface, `ICustomer` in this example.

Provide parameterization

That's a good start. But, the default implementation is too restrictive. Many consumers of this system may choose different thresholds for number of purchases, a different length of membership, or a different percentage discount. You can provide a better upgrade experience for more customers by providing a way to set those parameters. Let's add a static method that sets those three parameters controlling the default implementation:

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

There are many new language capabilities shown in that small code fragment. Interfaces can now include static members, including fields and methods. Different access modifiers are also enabled. The additional fields are private, the new method is public. Any of the modifiers are allowed on interface members.

Applications that use the general formula for computing the loyalty discount, but different parameters, don't need to provide a custom implementation; they can set the arguments through a static method. For example, the following code sets a "customer appreciation" that rewards any customer with more than one month's membership:

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Extend the default implementation

The code you've added so far has provided a convenient implementation for those scenarios where users want something like the default implementation, or to provide an unrelated set of rules. For a final feature, let's refactor the code a bit to enable scenarios where users may want to build on the default implementation.

Consider a startup that wants to attract new customers. They offer a 50% discount off a new customer's first order. Otherwise, existing customers get the standard discount. The library author needs to move the default implementation into a `protected static` method so that any class implementing this interface can reuse the code in their implementation. The default implementation of the interface member calls this shared method as well:

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

In an implementation of a class that implements this interface, the override can call the static helper method, and extend that logic to provide the "new customer" discount:

```
public decimal ComputeLoyaltyDiscount()
{
    if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

You can see the entire finished code in our [samples repo on GitHub](#). You can get the starter application on our [samples repo on GitHub](#).

These new features mean that interfaces can be updated safely when there's a reasonable default implementation for those new members. Carefully design interfaces to express single functional ideas that can be implemented by multiple classes. That makes it easier to upgrade those interface definitions when new requirements are discovered for that same functional idea.

Tutorial: Mix functionality in when creating classes using interfaces with default interface methods

3/23/2021 • 8 minutes to read • [Edit Online](#)

Beginning with C# 8.0 on .NET Core 3.0, you can define an implementation when you declare a member of an interface. This feature provides new capabilities where you can define default implementations for features declared in interfaces. Classes can pick when to override functionality, when to use the default functionality, and when not to declare support for discrete features.

In this tutorial, you'll learn how to:

- Create interfaces with implementations that describe discrete features.
- Create classes that use the default implementations.
- Create classes that override some or all of the default implementations.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8.0 compiler is available starting with [Visual Studio 2019 version 16.3](#), or the [.NET Core 3.0 SDK](#) or later.

Limitations of extension methods

One way you can implement behavior that appears as part of an interface is to define [extension methods](#) that provide the default behavior. Interfaces declare a minimum set of members while providing a greater surface area for any class that implements that interface. For example, the extension methods in [Enumerable](#) provide the implementation for any sequence to be the source of a LINQ query.

Extension methods are resolved at compile time, using the declared type of the variable. Classes that implement the interface can provide a better implementation for any extension method. Variable declarations must match the implementing type to enable the compiler to choose that implementation. When the compile-time type matches the interface, method calls resolve to the extension method. Another concern with extension methods is that those methods are accessible wherever the class containing the extension methods is accessible. Classes cannot declare if they should or should not provide features declared in extension methods.

Starting with C# 8.0, you can declare the default implementations as interface methods. Then, every class automatically uses the default implementation. Any class that can provide a better implementation can override the interface method definition with a better algorithm. In one sense, this technique sounds similar to how you could use [extension methods](#).

In this article, you'll learn how default interface implementations enable new scenarios.

Design the application

Consider a home automation application. You probably have many different types of lights and indicators that could be used throughout the house. Every light must support APIs to turn them on and off, and to report the current state. Some lights and indicators may support other features, such as:

- Turn light on, then turn it off after a timer.
- Blink the light for a period of time.

Some of these extended capabilities could be emulated in devices that support the minimal set. That indicates

providing a default implementation. For those devices that have more capabilities built in, the device software would use the native capabilities. For other lights, they could choose to implement the interface and use the default implementation.

Default interface members is a better solution for this scenario than extension methods. Class authors can control which interfaces they choose to implement. Those interfaces they choose are available as methods. In addition, because default interface methods are virtual by default, the method dispatch always chooses the implementation in the class.

Let's create the code to demonstrate these differences.

Create interfaces

Start by creating the interface that defines the behavior for all lights:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

A basic overhead light fixture might implement this interface as shown in the following code:

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {isOn ? "on" : "off"}";
}
```

In this tutorial, the code doesn't drive IoT devices, but emulates those activities by writing messages to the console. You can explore the code without automating your house.

Next, let's define the interface for a light that can automatically turn off after a timeout:

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

You could add a basic implementation to the overhead light, but a better solution is to modify this interface definition to provide a `virtual` default implementation:

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

By adding that change, the `OverheadLight` class can implement the timer function by declaring support for the interface:

```
public class OverheadLight : ITimerLight { }
```

A different light type may support a more sophisticated protocol. It can provide its own implementation for `TurnOnFor`, as shown in the following code:

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

Unlike overriding virtual class methods, the declaration of `TurnOnFor` in the `HalogenLight` class does not use the `override` keyword.

Mix and match capabilities

The advantages of default interface methods become clearer as you introduce more advanced capabilities. Using interfaces enables you to mix and match capabilities. It also enables each class author to choose between the default implementation and a custom implementation. Let's add an interface with a default implementation for a blinking light:

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for IBlinkingLight.Blink.");
    }
}
```

The default implementation enables any light to blink. The overhead light can add both timer and blink

capabilities using the default implementation:

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

A new light type, the `LEDLight` supports both the timer function and the blink function directly. This light style implements both the `ITimerLight` and `IBlinkingLight` interfaces, and overrides the `Blink` method:

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

An `ExtraFancyLight` might support both blink and timer functions directly:

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

The `HalogenLight` you created earlier doesn't support blinking. So, don't add the `IBlinkingLight` to the list of its supported interfaces.

Detect the light types using pattern matching

Next, let's write some test code. You can make use of C#'s [pattern matching](#) feature to determine a light's

capabilities by examining which interfaces it supports. The following method exercises the supported capabilities of each light:

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\\tAfter switching on, the light is {(light.IsOn() ? "on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\\tAfter switching off, the light is {(light.IsOn() ? "on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\\tBlink function not supported.");
    }
}
```

The following code in your `Main` method creates each light type in sequence and tests that light:

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

How the compiler determines best implementation

This scenario shows a base interface without any implementations. Adding a method into the `ILight` interface introduces new complexities. The language rules governing default interface methods minimize the effect on the

concrete classes that implement multiple derived interfaces. Let's enhance the original interface with a new method to show how that changes its use. Every indicator light can report its power status as an enumerated value:

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

The default implementation assumes no power:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

These changes compile cleanly, even though the `ExtraFancyLight` declares support for the `ILight` interface and both derived interfaces, `ITimerLight` and `IBlinkingLight`. There's only one "closest" implementation declared in the `ILight` interface. Any class that declared an override would become the one "closest" implementation. You saw examples in the preceding classes that overrode the members of other derived interfaces.

Avoid overriding the same method in multiple derived interfaces. Doing so creates an ambiguous method call whenever a class implements both derived interfaces. The compiler can't pick a single better method so it issues an error. For example, if both the `IBlinkingLight` and `ITimerLight` implemented an override of `PowerStatus`, the `OverheadLight` would need to provide a more specific override. Otherwise, the compiler can't pick between the implementations in the two derived interfaces. You can usually avoid this situation by keeping interface definitions small and focused on one feature. In this scenario, each capability of a light is its own interface; multiple interfaces are only inherited by classes.

This sample shows one scenario where you can define discrete features that can be mixed into classes. You declare any set of supported functionality by declaring which interfaces a class supports. The use of virtual default interface methods enables classes to use or define a different implementation for any or all the interface methods. This language capability provides new ways to model the real-world systems you're building. Default interface methods provide a clearer way to express related classes that may mix and match different features using virtual implementations of those capabilities.

Indices and ranges

3/23/2021 • 6 minutes to read • [Edit Online](#)

Ranges and indices provide a succinct syntax for accessing single elements or ranges in a sequence.

In this tutorial, you'll learn how to:

- Use the syntax for ranges in a sequence.
- Understand the design decisions for the start and end of each sequence.
- Learn scenarios for the [Index](#) and [Range](#) types.

Language support for indices and ranges

This language support relies on two new types and two new operators:

- [System.Index](#) represents an index into a sequence.
- The index from end operator `^`, which specifies that an index is relative to the end of a sequence.
- [System.Range](#) represents a sub range of a sequence.
- The range operator `...`, which specifies the start and end of a range as its operands.

Let's start with the rules for indices. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. The expression `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
    "The",           // index from start   index from end
    "quick",         // 0                  ^9
    "brown",         // 1                  ^8
    "fox",           // 2                  ^7
    "jumped",        // 3                  ^6
    "over",          // 4                  ^5
    "the",           // 5                  ^4
    "lazy",          // 6                  ^3
    "dog"            // 7                  ^2
};                // 8 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index. Add the following code below the initialization:

```
Console.WriteLine($"The last word is {words[^1]}");
```

A range specifies the *start* and *end* of a range. Ranges are exclusive, meaning the *end* isn't included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range. Add the following code to the same method. Copy and paste it at the bottom of the interactive window.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

The following code returns the range with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included. Add the following code as well:

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

The following examples create ranges that are open ended for the start, end, or both:

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

You can also declare ranges or indices as variables. The variable can then be used inside the `[` and `]` characters:

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

The following sample shows many of the reasons for those choices. Modify `x`, `y`, and `z` to try different combinations. When you experiment, use values where `x` is less than `y`, and `y` is less than `z` for valid combinations. Add the following code in a new method. Try different combinations:

```

int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\\tnumbers[x..y] is {x_y[0]} through {x_y[^1]}, numbers[y..z] is {y_z[0]} through
{y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\\tnumbers[x..^x] starts with {x_x[0]} and ends with {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\\t{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");

```

Type support for indices and ranges

Indexes and ranges provide clear, concise syntax to access a single element or a range of elements in a sequence. An index expression typically returns the type of the elements of a sequence. A range expression typically returns the same sequence type as the source sequence.

Any type that provides an [indexer](#) with an [Index](#) or [Range](#) parameter explicitly supports indices or ranges respectively. An indexer that takes a single [Range](#) parameter may return a different sequence type, such as [System.Span<T>](#).

IMPORTANT

The performance of code using the range operator depends on the type of the sequence operand.

The time complexity of the range operator depends on the sequence type. For example, if the sequence is a [string](#) or an array, then the result is a copy of the specified section of the input, so the time complexity is $O(N)$ (where N is the length of the range). On the other hand, if it's a [System.Span<T>](#) or a [System.Memory<T>](#), the result references the same backing store, which means there is no copy and the operation is $O(1)$.

In addition to the time complexity, this causes extra allocations and copies, impacting performance. In performance sensitive code, consider using [Span<T>](#) or [Memory<T>](#) as the sequence type, since the range operator does not allocate for them.

A type is **countable** if it has a property named [Length](#) or [Count](#) with an accessible getter and a return type of [int](#). A countable type that doesn't explicitly support indices or ranges may provide an implicit support for them. For more information, see the [Implicit Index support](#) and [Implicit Range support](#) sections of the [feature proposal note](#). Ranges using implicit range support return the same sequence type as the source sequence.

For example, the following .NET types support both indices and ranges: [String](#), [Span<T>](#), and [ReadOnlySpan<T>](#). The [List<T>](#) supports indices but doesn't support ranges.

[Array](#) has more nuanced behavior. Single dimension arrays support both indices and ranges. Multi-dimensional arrays don't support indexers or ranges. The indexer for a multi-dimensional array has multiple parameters, not

a single parameter. Jagged arrays, also referred to as an array of arrays, support both ranges and indexers. The following example shows how to iterate a rectangular subsection of a jagged array. It iterates the section in the center, excluding the first and last three rows, and the first and last two columns from each selected row:

```
var jagged = new int[10][]{  
    new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},  
    new int[10] { 10,11,12,13,14,15,16,17,18,19},  
    new int[10] { 20,21,22,23,24,25,26,27,28,29},  
    new int[10] { 30,31,32,33,34,35,36,37,38,39},  
    new int[10] { 40,41,42,43,44,45,46,47,48,49},  
    new int[10] { 50,51,52,53,54,55,56,57,58,59},  
    new int[10] { 60,61,62,63,64,65,66,67,68,69},  
    new int[10] { 70,71,72,73,74,75,76,77,78,79},  
    new int[10] { 80,81,82,83,84,85,86,87,88,89},  
    new int[10] { 90,91,92,93,94,95,96,97,98,99},  
};  
  
var selectedRows = jagged[3..^3];  
  
foreach (var row in selectedRows)  
{  
    var selectedColumns = row[2..^2];  
    foreach (var cell in selectedColumns)  
    {  
        Console.Write($"{cell}, ");  
    }  
    Console.WriteLine();  
}
```

In all cases, the range operator for [Array](#) allocates an array to store the elements returned.

Scenarios for indices and ranges

You'll often use ranges and indices when you want to analyze a portion of a larger sequence. The new syntax is clearer in reading exactly what portion of the sequence is involved. The local function [MovingAverage](#) takes a [Range](#) as its argument. The method then enumerates just that range when calculating the min, max, and average. Try the following code in your project:

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}: \tMin: {min},\tMax: {max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^start + 10..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}: \tMin: {min},\tMax: {max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
(
    subSequence[range].Min(),
    subSequence[range].Max(),
    subSequence[range].Average()
);

int[] Sequence(int count) =>
Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) * 100)).ToArray();
```

Tutorial: Express your design intent more clearly with nullable and non-nullable reference types

3/23/2021 • 10 minutes to read • [Edit Online](#)

C# 8.0 introduces [nullable reference types](#), which complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value*, others *may be missing a value*.

In this tutorial, you'll learn how to:

- Incorporate nullable and non-nullable reference types into your designs
- Enable nullable reference type checks throughout your code.
- Write code where the compiler enforces those design decisions.
- Use the nullable reference feature in your own designs

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8.0 compiler is available with [Visual Studio 2019](#), or [.NET Core 3.0](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Incorporate nullable reference types into your designs

In this tutorial, you'll build a library that models running a survey. The code uses both nullable reference types and non-nullable reference types to represent the real-world concepts. The survey questions can never be null. A respondent might prefer not to answer a question. The responses might be `null` in this case.

The code you'll write for this sample expresses that intent, and the compiler enforces that intent.

Create the application and enable nullable reference types

Create a new console application either in Visual Studio or from the command line using `dotnet new console`. Name the application `NullableIntroduction`. Once you've created the application, you'll need to specify that the entire project compiles in an enabled **nullable annotation context**. Open the `.csproj` file and add a `Nullable` element to the `PropertyGroup` element. Set its value to `enable`. You must opt into the **nullable reference types** feature, even in C# 8.0 projects. That's because once the feature is turned on, existing reference variable declarations become **non-nullable reference types**. While that decision will help find issues where existing code may not have proper null-checks, it may not accurately reflect your original design intent:

```
<Nullable>enable</Nullable>
```

Design the types for the application

This survey application requires creating a number of classes:

- A class that models the list of questions.
- A class that models a list of people contacted for the survey.
- A class that models the answers from a person that took the survey.

These types will make use of both nullable and non-nullable reference types to express which members are required and which members are optional. Nullable reference types communicate that design intent clearly:

- The questions that are part of the survey can never be null: It makes no sense to ask an empty question.
- The respondents can never be null. You'll want to track people you contacted, even respondents that declined to participate.
- Any response to a question may be null. Respondents can decline to answer some or all questions.

If you've programmed in C#, you may be so accustomed to reference types that allow `null` values that you may have missed other opportunities to declare non-nullable instances:

- The collection of questions should be non-nullable.
- The collection of respondents should be non-nullable.

As you write the code, you'll see that a non-nullable reference type as the default for references avoids common mistakes that could lead to [NullReferenceExceptions](#). One lesson from this tutorial is that you made decisions about which variables could or could not be `null`. The language didn't provide syntax to express those decisions. Now it does.

The app you'll build does the following steps:

1. Creates a survey and adds questions to it.
2. Creates a pseudo-random set of respondents for the survey.
3. Contacts respondents until the completed survey size reaches the goal number.
4. Writes out important statistics on the survey responses.

Build the survey with nullable and non-nullable reference types

The first code you'll write creates the survey. You'll write classes to model a survey question and a survey run. Your survey has three types of questions, distinguished by the format of the answer: Yes/No answers, number answers, and text answers. Create a `public SurveyQuestion` class:

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

The compiler interprets every reference type variable declaration as a **non-nullable** reference type for code in an enabled nullable annotation context. You can see your first warning by adding properties for the question text and the type of question, as shown in the following code:

```

namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}

```

Because you haven't initialized `QuestionText`, the compiler issues a warning that a non-nullable property hasn't been initialized. Your design requires the question text to be non-null, so you add a constructor to initialize it and the `QuestionType` value as well. The finished class definition looks like the following code:

```

namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }

        public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
            (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
    }
}

```

Adding the constructor removes the warning. The constructor argument is also a non-nullable reference type, so the compiler doesn't issue any warnings.

Next, create a `public` class named `SurveyRun`. This class contains a list of `SurveyQuestion` objects and methods to add questions to the survey, as shown in the following code:

```

using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) => surveyQuestions.Add(surveyQuestion);
    }
}

```

As before, you must initialize the list object to a non-null value or the compiler issues a warning. There are no null checks in the second overload of `AddQuestion` because they aren't needed: You've declared that variable to

be non-nullable. Its value can't be `null`.

Switch to *Program.cs* in your editor and replace the contents of `Main` with the following lines of code:

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the nearest 100) has that
happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Because the entire project is in an enabled nullable annotation context, you'll get warnings when you pass `null` to any method expecting a non-nullable reference type. Try it by adding the following line to `Main`:

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Create respondents and get answers to the survey

Next, write the code that generates answers to the survey. This process involves several small tasks:

1. Build a method that generates respondent objects. These represent people asked to fill out the survey.
2. Build logic to simulate asking the questions to a respondent and collecting answers or noting that a respondent didn't answer.
3. Repeat until enough respondents have answered the survey.

You'll need a class to represent a survey response, so add that now. Enable nullable support. Add an `Id` property and a constructor that initializes it, as shown in the following code:

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Next, add a `static` method to create new participants by generating a random ID:

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new SurveyResponse(randomGenerator.Next());
```

The main responsibility of this class is to generate the responses for a participant to the questions in the survey. This responsibility has a few steps:

1. Ask for participation in the survey. If the person doesn't consent, return a missing (or null) response.
2. Ask each question and record the answer. Each answer may also be missing (or null).

Add the following code to your `SurveyResponse` class:

```

private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}

```

The storage for the survey answers is a `Dictionary<int, string>?`, indicating that it may be null. You're using the new language feature to declare your design intent, both to the compiler and to anyone reading your code later. If you ever dereference `surveyResponses` without checking for the `null` value first, you'll get a compiler warning. You don't get a warning in the `AnswerSurvey` method because the compiler can determine the `surveyResponses` variable was set to a non-null value above.

Using `null` for missing answers highlights a key point for working with nullable reference types: your goal isn't to remove all `null` values from your program. Rather, your goal is to ensure that the code you write expresses the intent of your design. Missing values are a necessary concept to express in your code. The `null` value is a clear way to express those missing values. Trying to remove all `null` values only leads to defining some other way to express those missing values without `null`.

Next, you need to write the `PerformSurvey` method in the `SurveyRun` class. Add the following code in the `SurveyRun` class:

```

private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}

```

Here again, your choice of a nullable `List<SurveyResponse>?` indicates the response may be null. That indicates the survey hasn't been given to any respondents yet. Notice that respondents are added until enough have consented.

The last step to run the survey is to add a call to perform the survey at the end of the `Main` method:

```
surveyRun.PerformSurvey(50);
```

Examine survey responses

The last step is to display survey results. You'll add code to many of the classes you've written. This code demonstrates the value of distinguishing nullable and non-nullable reference types. Start by adding the following two expression-bodied members to the `SurveyResponse` class:

```

public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No answer";

```

Because `surveyResponses` is a nullable reference type, null checks are necessary before de-referencing it. The `Answer` method returns a non-nullable string, so we have to cover the case of a missing answer by using the null-coalescing operator.

Next, add these three expression-bodied members to the `SurveyRun` class:

```

public IEnumerable<SurveyResponse> AllParticipants => (respondents ?? Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];

```

The `AllParticipants` member must take into account that the `respondents` variable might be null, but the return value can't be null. If you change that expression by removing the `??` and the empty sequence that follows, the compiler warns you the method might return `null` and its return signature returns a non-nullable type.

Finally, add the following loop at the bottom of the `Main` method:

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"{surveyRun.GetQuestion(i).QuestionText} : {answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

You don't need any `null` checks in this code because you've designed the underlying interfaces so that they all return non-nullable reference types.

Get the code

You can get the code for the finished tutorial from our [samples](#) repository in the `csharp/NullableIntroduction` folder.

Experiment by changing the type declarations between nullable and non-nullable reference types. See how that generates different warnings to ensure you don't accidentally dereference a `null`.

Next steps

Learn more by migrating an existing application to use nullable reference types:

[Upgrade an application to use nullable reference types](#)

Learn how to use nullable reference type when using Entity Framework:

[Entity Framework Core Fundamentals: Working with Nullable Reference Types](#)

Tutorial: Migrate existing code with nullable reference types

3/23/2021 • 13 minutes to read • [Edit Online](#)

C# 8 introduces **nullable reference types**, which complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value*, others *may be missing a value*. Any existing variables of a reference type would be interpreted as a non-nullable reference type.

In this tutorial, you'll learn how to:

- Enable null reference checks as you work with code.
- Diagnose and correct different warnings related to null values.
- Manage the interface between nullable enabled and nullable disabled contexts.
- Control nullable annotation contexts.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8 compiler is available starting with [Visual Studio 2019 version 16.3](#) or [.NET Core 3.0 SDK](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Explore the sample application

The sample application that you'll migrate is an RSS feed reader web app. It reads from a single RSS feed and displays summaries for the most recent articles. You can select any of the articles to visit the site. The application is relatively new but was written before nullable reference types were available. The design decisions for the application represented sound principles, but don't take advantage of this important language feature.

The sample application includes a unit test library that validates the major functionality of the app. That project will make it easier to upgrade safely, if you change any of the implementation based on the warnings generated. You can download the starter code from the [dotnet/samples](#) GitHub repository.

Your goal migrating a project should be to leverage the new language features so that you clearly express your intent on the nullability of variables, and do so in such a way that the compiler doesn't generate warnings when you have the nullable annotation context and nullable warning context set to `enabled`.

Upgrade the projects to C# 8

A good first step is to determine the scope of the migration task. Start by upgrading the project to C# 8.0 (or newer). Add the `LangVersion` element to the PropertyGroup in both csproj files for the web project and the unit test project:

```
<LangVersion>8.0</LangVersion>
```

Upgrading the language version selects C# 8.0, but does not enable the nullable annotation context or the nullable warning context. Rebuild the project to ensure that it builds without warnings.

A good next step is to turn on the nullable annotation context and see how many warnings are generated. Add the following element to both csproj files in the solution, directly under the `LangVersion` element:

```
<Nullable>enable</Nullable>
```

Do a test build, and notice the warning list. In this small application, the compiler generates five warnings, so it's likely you'd leave the nullable annotation context enabled and start fixing warnings for the entire project.

That strategy works only for smaller projects. For any larger projects, the number of warnings generated by enabling the nullable annotation context for the entire codebase makes it harder to fix the warnings systematically. For larger enterprise projects, you'll often want to migrate one project at a time. In each project, migrate one class or file at a time.

Warnings help discover original design intent

There are two classes that generate multiple warnings. Start with the `NewsStoryViewModel` class. Remove the `Nullable` element from both csproj files so that you can limit the scope of warnings to the sections of code you're working with. Open the `NewsStoryViewModel.cs` file and add the following directives to enable the nullable annotation context for the `NewsStoryViewModel` and restore it following that class definition:

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
#nullable restore
```

These two directives help you focus your migration efforts. The nullable warnings are generated for the area of code you're actively working on. You'll leave them on until you're ready to turn on the warnings for the entire project. You should use the `restore` rather than `disable` value so that you don't accidentally disable the context later when you've turned on nullable annotations for the entire project. Once you've turned on the nullable annotation context for the entire project, you can remove all the `#nullable` pragmas from that project.

The `NewsStoryViewModel` class is a data transfer object (DTO) and two of the properties are read/write strings:

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
```

These two properties cause `CS8618`, "Non-nullable property is uninitialized". That's clear enough: both `string` properties have the default value of `null` when a `NewsStoryViewModel` is constructed. What's important to discover is how `NewsStoryViewModel` objects are constructed. Looking at this class, you can't tell if the `null` value is part of the design, or if these objects are set to non-null values whenever one is created. The news stories are created in the `GetNews` method of the `NewsService` class:

```
ISyndicationItem item = await feedReader.ReadItem();
var newsStory = _mapper.Map<NewsStoryViewModel>(item);
news.Add(newsStory);
```

There's quite a bit going on in the preceding block of code. This application uses the [AutoMapper](#) NuGet package to construct a news item from an `ISyndicationItem`. You've discovered that the news story items are constructed and the properties are set in that one statement. That means the design for the `NewsStoryViewModel` indicates that these properties should never have the `null` value. These properties should be **nonnullable reference types**. That best expresses the original design intent. In fact, any `NewsStoryViewModel` is correctly instantiated with non-null values. That makes the following initialization code a valid fix:

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; } = default!;
    public string Uri { get; set; } = default!;
}
```

The assignment of `Title` and `Uri` to `default` which is `null` for the `string` type doesn't change the runtime behavior of the program. The `NewsStoryViewModel` is still constructed with null values, but now the compiler reports no warnings. The **null-forgiving operator**, the `!` character following the `default` expression tells the compiler that the preceding expression is not null. This technique may be expedient when other changes force much larger changes to a code base, but in this application there is a relatively quick and better solution: Make the `NewsStoryViewModel` an immutable type where all the properties are set in the constructor. Make the following changes to the `NewsStoryViewModel`:

```
#nullable enable
public class NewsStoryViewModel
{
    public NewsStoryViewModel(DateTimeOffset published, string title, string uri) =>
        (Published, Title, Uri) = (published, title, uri);

    public DateTimeOffset Published { get; }
    public string Title { get; }
    public string Uri { get; }
}
#nullable restore
```

Once that's done, you need to update the code that configures the AutoMapper so that it uses the constructor rather than setting properties. Open `NewsService.cs` and look for the following code at the bottom of the file:

```
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForMember(dest => dest.Uri, opts => opts.MapFrom(src => src.Id));
    }
}
```

That code maps properties of the `ISyndicationItem` object to the `NewsStoryViewModel` properties. You want the AutoMapper to provide the mapping using a constructor instead. Replace the above code with the following automapper configuration:

```

#nullable enable
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForCtorParam("uri", opt => opt.MapFrom(src => src.Id));
    }
}

```

Notice that because this class is small, and you've examined carefully, you should turn on the `#nullable enable` directive above this class declaration. The change to the constructor could have broken something, so it's worthwhile to run all the tests and test the application before moving on.

The first set of changes showed you how to discover when the original design indicated that variables shouldn't be set to `null`. The technique is referred to as **correct by construction**. You declare that an object and its properties cannot be `null` when it's constructed. The compiler's flow analysis provides assurance that those properties aren't set to `null` after construction. Note that this constructor is called by external code, and that code is **nullable oblivious**. The new syntax doesn't provide runtime checking. External code might circumvent the compiler's flow analysis.

Other times, the structure of a class provides different clues to the intent. Open the `Error.cshtml.cs` file in the `Pages` folder. The `ErrorViewModel` contains the following code:

```

public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}

```

Add the `#nullable enable` directive before the class declaration, and a `#nullable restore` directive after it. You'll get one warning that `RequestId` is not initialized. By looking at the class, you should decide that the `RequestId` property should be null in some cases. The existence of the `ShowRequestId` property indicates that missing values are possible. Because `null` is valid, add the `?` on the `string` type to indicate the `RequestId` property is a *nullable reference type*. The final class looks like the following example:

```

#nullable enable
public class ErrorModel : PageModel
{
    public string? RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
#nullable restore

```

Check for the uses of the property, and you see that in the associated page, the property is checked for null

before rendering it in markup. That's a safe use of a nullable reference type, so you're done with this class.

Fixing nulls causes change

Frequently, the fix for one set of warnings creates new warnings in related code. Let's see the warnings in action by fixing the `index.cshtml.cs` class. Open the `index.cshtml.cs` file and examine the code. This file contains the code behind for the index page:

```
public class IndexModel : PageModel
{
    private readonly NewsService _newsService;

    public IndexModel(NewsService newsService)
    {
        _newsService = newsService;
    }

    public string ErrorText { get; private set; }

    public List<NewsStoryViewModel> NewsItems { get; private set; }

    public async Task OnGet()
    {
        string feedUrl = Request.Query["feedurl"];

        if (!string.IsNullOrEmpty(feedUrl))
        {
            try
            {
                NewsItems = await _newsService.GetNews(feedUrl);
            }
            catch (UriFormatException)
            {
                ErrorText = "There was a problem parsing the URL.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.NameResolutionFailure)
            {
                ErrorText = "Unknown host name.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
            {
                ErrorText = "Syndication feed not found.";
                return;
            }
            catch (AggregateException ae)
            {
                ae.Handle((x) =>
                {
                    if (x is XmlException)
                    {
                        ErrorText = "There was a problem parsing the feed. Are you sure that URL is a syndication feed?";
                        return true;
                    }
                    return false;
                });
            }
        }
    }
}
```

Add the `#nullable enable` directive and you'll see two warnings. Neither the `ErrorText` property nor the

`NewsItems` property is initialized. An examination of this class would lead you to believe that both properties should be nullable reference types: Both have private setters. Exactly one is assigned in the `OnGet` method. Before making changes, look at the consumers of both properties. In the page itself, the `ErrorText` is checked against null before generating markup for any errors. The `NewsItems` collection is checked against `null`, and checked to ensure the collection has items. A quick fix would be to make both properties nullable reference types. A better fix would be to make the collection a nonnullable reference type, and add items to the existing collection when retrieving news. The first fix is to add the `?` to the `string` type for the `ErrorText`:

```
public string? ErrorText { get; private set; }
```

That change won't ripple through other code, because any access to the `ErrorText` property was already guarded by null checks. Next, initialize the `NewsItems` list and remove the property setter, making it a readonly property:

```
public List<NewsStoryViewModel> NewsItems { get; } = new List<NewsStoryViewModel>();
```

That fixed the warning but introduced an error. The `NewsItems` list is now **correct by construction**, but the code that sets the list in `OnGet` must change to match the new API. Instead of an assignment, call `AddRange` to add the news items to the existing list:

```
NewsItems.AddRange(await _newsService.GetNews(feedUrl));
```

Using `AddRange` instead of an assignment means that the `GetNews` method can return an `IEnumerable` instead of a `List`. That saves one allocation. Change the signature of the method, and remove the `ToList` call, as shown in the following code sample:

```

public async Task<IEnumerable<NewsStoryViewModel>> GetNews(string feedUrl)
{
    var news = new List<NewsStoryViewModel>();
    var feedUri = new Uri(feedUrl);

    using (var xmlReader = XmlReader.Create(feedUri.ToString(),
        new XmlReaderSettings { Async = true }))
    {
        try
        {
            var feedReader = new RssFeedReader(xmlReader);

            while (await feedReader.Read())
            {
                switch (feedReader.ElementType)
                {
                    // RSS Item
                    case SyndicationElementType.Item:
                        ISyndicationItem item = await feedReader.ReadItem();
                        var newsStory = _mapper.Map<NewsStoryViewModel>(item);
                        news.Add(newsStory);
                        break;

                    // Something else
                    default:
                        break;
                }
            }
        }
        catch (AggregateException ae)
        {
            throw ae.Flatten();
        }
    }

    return news.OrderByDescending(story => story.Published);
}

```

Changing the signature breaks one of tests as well. Open the `NewsServiceTests.cs` file in the `Services` folder of the `SimpleFeedReader.Tests` project. Navigate to the `Returns_News_Stories_Given_Valid_Url` test and change the type of the `result` variable to `IEnumerable<NewsItem>`. Changing the type means the `Count` property is no longer available, so replace the `Count` property in the `Assert` with a call to `Any()`:

```

// Act
IEnumerable<NewsStoryViewModel> result =
    await _newsService.GetNews(feedUrl);

// Assert
Assert.True(result.Any());

```

You'll need to add a `using System.Linq` statement to the beginning of the file as well.

This set of changes highlights special consideration when updating code that includes generic instantiations. Both the list and the elements in the list of non-nullable types. Either or both could be nullable types. All the following declarations are allowed:

- `List<NewsStoryViewModel>` : nonnullable list of nonnullable view models.
- `List<NewsStoryViewModel?>` : nonnullable list of nullable view models.
- `List<NewsStoryViewModel?>` : nullable list of nonnullable view models.
- `List<NewsStoryViewModel?>?` : nullable list of nullable view models.

Interfaces with external code

You've made changes to the `NewsService` class, so turn on the `#nullable enable` annotation for that class. This won't generate any new warnings. However, careful examination of the class helps to illustrate some of the limitations of the compiler's flow analysis. Examine the constructor:

```
public NewsService(IMapper mapper)
{
    _mapper = mapper;
}
```

The `IMapper` parameter is typed as a nonnullable reference. It's called by ASP.NET Core infrastructure code, so the compiler doesn't really know that the `IMapper` will never be null. The default ASP.NET Core dependency injection (DI) container throws an exception if it can't resolve a necessary service, so the code is correct. The compiler can't validate all calls to your public APIs, even if your code is compiled with nullable annotation contexts enabled. Furthermore, your libraries may be consumed by projects that have not yet opted into using nullable reference types. Validate inputs to public APIs even though you've declared them as nonnullable types.

Get the code

You've fixed the warnings you identified in the initial test compile, so now you can turn on the nullable annotation context for both projects. Rebuild the projects; the compiler reports no warnings. You can get the code for the finished project in the [dotnet/samples](#) GitHub repository.

The new features that support nullable reference types help you find and fix potential errors in how you handle `null` values in your code. Enabling the nullable annotation context allows you to express your design intent: some variables should never be null, other variables may contain null values. These features make it easier for you to declare your design intent. Similarly, the nullable warning context instructs the compiler to issue warnings when you have violated that intent. Those warnings guide you to make updates that make your code more resilient and less likely to throw a `NullReferenceException` during execution. You can control the scope of these contexts so that you can focus on local areas of code to migrate while the remaining codebase is untouched. In practice, you can make this migration task a part of regular maintenance to your classes. This tutorial demonstrated the process to migrate an application to use nullable reference types. You can explore a larger real-world example of this process by examining the PR [Jon Skeet](#) made to incorporate nullable reference types into [NodaTime](#). Or just in addition, you can learn techniques for using nullable reference types with Entity Framework Core in [Entity Framework Core - Working with nullable reference types](#).

Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0

4/6/2021 • 9 minutes to read • [Edit Online](#)

C# 8.0 introduces **async streams**, which model a streaming source of data. Data streams often retrieve or generate elements asynchronously. Async streams rely on new interfaces introduced in .NET Standard 2.1. These interfaces are supported in .NET Core 3.0 and later. They provide a natural programming model for asynchronous streaming data sources.

In this tutorial, you'll learn how to:

- Create a data source that generates a sequence of data elements asynchronously.
- Consume that data source asynchronously.
- Support cancellation and captured contexts for asynchronous streams.
- Recognize when the new interface and data source are preferred to earlier synchronous data sequences.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8 compiler is available starting with [Visual Studio 2019 version 16.3](#) or [.NET Core 3.0 SDK](#).

You'll need to create a [GitHub access token](#) so that you can access the GitHub GraphQL endpoint. Select the following permissions for your GitHub Access Token:

- repo:status
- public_repo

Save the access token in a safe place so you can use it to gain access to the GitHub API endpoint.

WARNING

Keep your personal access token secure. Any software with your personal access token could make GitHub API calls using your access rights.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Run the starter application

You can get the code for the starter application used in this tutorial from the [dotnet/docs](#) repository in the [csharp/whats-new/tutorials](#) folder.

The starter application is a console application that uses the [GitHub GraphQL](#) interface to retrieve recent issues written in the [dotnet/docs](#) repository. Start by looking at the following code for the starter app `Main` method:

```

static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-
    token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
        "You must store your GitHub key in the 'GitHubKey' environment variable",
        "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}

```

You can either set a `GitHubKey` environment variable to your personal access token, or you can replace the last argument in the call to `GetEnvVariable` with your personal access token. Don't put your access code in source code if you'll be sharing the source with others. Never upload access codes to a shared source repository.

After creating the GitHub client, the code in `Main` creates a progress reporting object and a cancellation token. Once those objects are created, `Main` calls `runPagedQueryAsync` to retrieve the most recent 250 created issues. After that task has finished, the results are displayed.

When you run the starter application, you can make some important observations about how this application runs. You'll see progress reported for each page returned from GitHub. You can observe a noticeable pause before GitHub returns each new page of issues. Finally, the issues are displayed only after all 10 pages have been retrieved from GitHub.

Examine the implementation

The implementation reveals why you observed the behavior discussed in the previous section. Examine the code for `runPagedQueryAsync`:

```

private static async Task<JArray> runPagedQueryAsync(GitHubClient client, string queryText, string repoName,
CancellationToken cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();
        finalResults.Merge(issues(results)["nodes"]);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;
}

JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}

```

Let's concentrate on the paging algorithm and async structure of the preceding code. (You can consult the [GitHub GraphQL documentation](#) for details on the GitHub GraphQL API.) The `runPagedQueryAsync` method enumerates the issues from most recent to oldest. It requests 25 issues per page and examines the `pageInfo` structure of the response to continue with the previous page. That follows GraphQL's standard paging support for multi-page responses. The response includes a `pageInfo` object that includes a `hasPreviousPages` value and a `startCursor` value used to request the previous page. The issues are in the `nodes` array. The `runPagedQueryAsync` method appends these nodes to an array that contains all the results from all pages.

After retrieving and restoring a page of results, `runPagedQueryAsync` reports progress and checks for cancellation. If cancellation has been requested, `runPagedQueryAsync` throws an [OperationCanceledException](#).

There are several elements in this code that can be improved. Most importantly, `runPagedQueryAsync` must allocate storage for all the issues returned. This sample stops at 250 issues because retrieving all open issues would require much more memory to store all the retrieved issues. The protocols for supporting progress reports and cancellation make the algorithm harder to understand on its first reading. More types and APIs are involved. You must trace the communications through the `CancellationTokenSource` and its associated `CancellationToken` to understand where cancellation is requested and where it's granted.

Async streams provide a better way

Async streams and the associated language support address all those concerns. The code that generates the sequence can now use `yield return` to return elements in a method that was declared with the `async` modifier. You can consume an async stream using an `await foreach` loop just as you consume any sequence using a

`foreach` loop.

These new language features depend on three new interfaces added to .NET Standard 2.1 and implemented in .NET Core 3.0:

- [System.Collections.Generic.IAsyncEnumerable<T>](#)
- [System.Collections.Generic.IAsyncEnumerator<T>](#)
- [System.IAsyncDisposable](#)

These three interfaces should be familiar to most C# developers. They behave in a manner similar to their synchronous counterparts:

- [System.Collections.Generic.IEnumerable<T>](#)
- [System.Collections.Generic.IEnumerator<T>](#)
- [System.IDisposable](#)

One type that may be unfamiliar is [System.Threading.Tasks.ValueTask](#). The `ValueTask` struct provides a similar API to the [System.Threading.Tasks.Task](#) class. `ValueTask` is used in these interfaces for performance reasons.

Convert to async streams

Next, convert the `runPagedQueryAsync` method to generate an async stream. First, change the signature of `runPagedQueryAsync` to return an `IAsyncEnumerable<JToken>`, and remove the cancellation token and progress objects from the parameter list as shown in the following code:

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

The starter code processes each page as the page is retrieved, as shown in the following code:

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Replace those three lines with the following code:

```
foreach (JObject issue in issues(results)["nodes"])
    yield return issue;
```

You can also remove the declaration of `finalResults` earlier in this method and the `return` statement that follows the loop you modified.

You've finished the changes to generate an async stream. The finished method should resemble the following code:

```

private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}

```

Next, you change the code that consumes the collection to consume the async stream. Find the following code in `Main` that processes the collection of issues:

```

var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}

```

Replace that code with the following `await foreach` loop:

```
int num = 0;
await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

The new interface [IAsyncEnumerator<T>](#) derives from [IAsyncDisposable](#). That means the preceding loop will asynchronously dispose the stream when the loop finishes. You can imagine the loop looks like the following code:

```
int num = 0;
var enumerator = runPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the [TaskAsyncEnumerableExtensions.ConfigureAwait](#) extension method. For more information about synchronization contexts and capturing the current context, see the article on [consuming the Task-based asynchronous pattern](#).

Async streams support cancellation using the same protocol as other `async` methods. You would modify the signature for the async iterator method as follows to support cancellation:

```

private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation] CancellationToken cancellationToken =
default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}

```

The [EnumeratorCancellationAttribute](#) attribute causes the compiler to generate code for the [IAsyncEnumerable<T>](#) that makes the token passed to `GetAsyncEnumerator` visible to the body of the async iterator as that argument. Inside `runQueryAsync`, you could examine the state of the token and cancel further work if requested.

You use another extension method, [WithCancellation](#), to pass the cancellation token to the async stream. You would modify the loop enumerating the issues as follows:

```

private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}

```

You can get the code for the finished tutorial from the [dotnet/docs](#) repository in the [csharp/whats-new/tutorials](#) folder.

Run the finished application

Run the application again. Contrast its behavior with the behavior of the starter application. The first page of

results is enumerated as soon as it's available. There's an observable pause as each new page is requested and retrieved, then the next page's results are quickly enumerated. The `try` / `catch` block isn't needed to handle cancellation: the caller can stop enumerating the collection. Progress is clearly reported because the async stream generates results as each page is downloaded. The status for each issue returned is seamlessly included in the `await foreach` loop. You don't need a callback object to track progress.

You can see improvements in memory use by examining the code. You no longer need to allocate a collection to store all the results before they're enumerated. The caller can determine how to consume the results and if a storage collection is needed.

Run both the starter and finished applications and you can observe the differences between the implementations for yourself. You can delete the GitHub access token you created when you started this tutorial after you've finished. If an attacker gained access to that token, they could access GitHub APIs using your credentials.

Explore object oriented programming with classes and objects

3/6/2021 • 9 minutes to read • [Edit Online](#)

In this tutorial, you'll build a console application and see the basic object-oriented features that are part of the C# language.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

Create your application

Using a terminal window, create a directory named *classes*. You'll build your application there. Change to that directory and type `dotnet new console` in the console window. This command creates your application. Open *Program.cs*. It should look like this:

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

In this tutorial, you're going to create new types that represent a bank account. Typically developers define each class in a different text file. That makes it easier to manage as a program grows in size. Create a new file named *BankAccount.cs* in the *classes* directory.

This file will contain the definition of a *bank account*. Object Oriented programming organizes code by creating types in the form of *classes*. These classes contain the code that represents a specific entity. The `BankAccount` class represents a bank account. The code implements specific operations through methods and properties. In this tutorial, the bank account supports this behavior:

1. It has a 10-digit number that uniquely identifies the bank account.
2. It has a string that stores the name or names of the owners.
3. The balance can be retrieved.
4. It accepts deposits.
5. It accepts withdrawals.
6. The initial balance must be positive.
7. Withdrawals cannot result in a negative balance.

Define the bank account type

You can start by creating the basics of a class that defines that behavior. Create a new file using the **File>New** command. Name it *BankAccount.cs*. Add the following code to your *BankAccount.cs* file:

```
using System;

namespace classes
{
    public class BankAccount
    {
        public string Number { get; }
        public string Owner { get; set; }
        public decimal Balance { get; }

        public void MakeDeposit(decimal amount, DateTime date, string note)
        {
        }

        public void MakeWithdrawal(decimal amount, DateTime date, string note)
        {
        }
    }
}
```

Before going on, let's take a look at what you've built. The `namespace` declaration provides a way to logically organize your code. This tutorial is relatively small, so you'll put all the code in one namespace.

`public class BankAccount` defines the class, or type, you are creating. Everything inside the `{` and `}` that follows the class declaration defines the state and behavior of the class. There are five **members** of the `BankAccount` class. The first three are **properties**. Properties are data elements and can have code that enforces validation or other rules. The last two are **methods**. Methods are blocks of code that perform a single function. Reading the names of each of the members should provide enough information for you or another developer to understand what the class does.

Open a new account

The first feature to implement is to open a bank account. When a customer opens an account, they must supply an initial balance, and information about the owner or owners of that account.

Creating a new object of the `BankAccount` type means defining a **constructor** that assigns those values. A **constructor** is a member that has the same name as the class. It is used to initialize objects of that class type. Add the following constructor to the `BankAccount` type. Place the following code above the declaration of `MakeDeposit`:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Constructors are called when you create an object using `new`. Replace the line `Console.WriteLine("Hello World!");` in *Program.cs* with the following code (replace `<name>` with your name):

```
var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with {account.Balance} initial
balance.");
```

Let's run what you've built so far. If you're using Visual Studio, Select **Start without debugging** from the **Run**

menu. If you're using a command line, type `dotnet run` in the directory where you've created your project.

Did you notice that the account number is blank? It's time to fix that. The account number should be assigned when the object is constructed. But it shouldn't be the responsibility of the caller to create it. The `BankAccount` class code should know how to assign new account numbers. A simple way to do this is to start with a 10-digit number. Increment it when each new account is created. Finally, store the current account number when an object is constructed.

Add a member declaration to the `BankAccount` class. Place the following line of code after the opening brace `{` at the beginning of the `BankAccount` class:

```
private static int accountNumberSeed = 1234567890;
```

This is a data member. It's `private`, which means it can only be accessed by code inside the `BankAccount` class. It's a way of separating the public responsibilities (like having an account number) from the private implementation (how account numbers are generated). It is also `static`, which means it is shared by all of the `BankAccount` objects. The value of a non-static variable is unique to each instance of the `BankAccount` object. Add the following two lines to the constructor to assign the account number. Place them after the line that says `this.Balance = initialBalance`:

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Type `dotnet run` to see the results.

Create deposits and withdrawals

Your bank account class needs to accept deposits and withdrawals to work correctly. Let's implement deposits and withdrawals by creating a journal of every transaction for the account. That has a few advantages over simply updating the balance on each transaction. The history can be used to audit all transactions and manage daily balances. By computing the balance from the history of all transactions when needed, any errors in a single transaction that are fixed will be correctly reflected in the balance on the next computation.

Let's start by creating a new type to represent a transaction. This is a simple type that doesn't have any responsibilities. It needs a few properties. Create a new file named `Transaction.cs`. Add the following code to it:

```
using System;

namespace classes
{
    public class Transaction
    {
        public decimal Amount { get; }
        public DateTime Date { get; }
        public string Notes { get; }

        public Transaction(decimal amount, DateTime date, string note)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}
```

Now, let's add a `List<T>` of `Transaction` objects to the `BankAccount` class. Add the following declaration after

the constructor in your *BankAccount.cs* file:

```
private List<Transaction> allTransactions = new List<Transaction>();
```

The `List<T>` class requires you to import a different namespace. Add the following at the beginning of *BankAccount.cs*:

```
using System.Collections.Generic;
```

Now, let's correctly compute the `Balance`. The current balance can be found by summing the values of all transactions. As the code is currently, you can only get the initial balance of the account, so you'll have to update the `Balance` property. Replace the line `public decimal Balance { get; }` in *BankAccount.cs* with the following code:

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

This example shows an important aspect of *properties*. You're now computing the balance when another programmer asks for the value. Your computation enumerates all transactions, and provides the sum as the current balance.

Next, implement the `MakeDeposit` and `MakeWithdrawal` methods. These methods will enforce the final two rules: that the initial balance must be positive, and that any withdrawal must not create a negative balance.

This introduces the concept of *exceptions*. The standard way of indicating that a method cannot complete its work successfully is to throw an exception. The type of exception and the message associated with it describe the error. Here, the `MakeDeposit` method throws an exception if the amount of the deposit is negative. The `MakeWithdrawal` method throws an exception if the withdrawal amount is negative, or if applying the withdrawal results in a negative balance. Add the following code after the declaration of the `allTransactions` list:

```

public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}

```

The `throw` statement **throws** an exception. Execution of the current block ends, and control transfers to the first matching `catch` block found in the call stack. You'll add a `catch` block to test this code a little later on.

The constructor should get one change so that it adds an initial transaction, rather than updating the balance directly. Since you already wrote the `MakeDeposit` method, call it from your constructor. The finished constructor should look like this:

```

public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}

```

`DateTime.Now` is a property that returns the current date and time. Test this by adding a few deposits and withdrawals in your `Main` method, following the code that creates a new `BankAccount`:

```

account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);

```

Next, test that you are catching error conditions by trying to create an account with a negative balance. Add the following code after the preceding code you just added:

```
// Test that the initial balances must be positive.  
try  
{  
    var invalidAccount = new BankAccount("invalid", -55);  
}  
catch (ArgumentOutOfRangeException e)  
{  
    Console.WriteLine("Exception caught creating account with negative balance");  
    Console.WriteLine(e.ToString());  
}
```

You use the `try` and `catch` statements to mark a block of code that may throw exceptions and to catch those errors that you expect. You can use the same technique to test the code that throws an exception for a negative balance. Add the following code at the end of your `Main` method:

```
// Test for a negative balance.  
try  
{  
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");  
}  
catch (InvalidOperationException e)  
{  
    Console.WriteLine("Exception caught trying to overdraw");  
    Console.WriteLine(e.ToString());  
}
```

Save the file and type `dotnet run` to try it.

Challenge - log all transactions

To finish this tutorial, you can write the `GetAccountHistory` method that creates a `string` for the transaction history. Add this method to the `BankAccount` type:

```
public string GetAccountHistory()  
{  
    var report = new System.Text.StringBuilder();  
  
    decimal balance = 0;  
    report.AppendLine("Date\tAmount\tBalance\tNote");  
    foreach (var item in allTransactions)  
    {  
        balance += item.Amount;  
        report.AppendLine($"{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");  
    }  
  
    return report.ToString();  
}
```

This uses the `StringBuilder` class to format a string that contains one line for each transaction. You've seen the string formatting code earlier in these tutorials. One new character is `\t`. That inserts a tab to format the output.

Add this line to test it in `Program.cs`:

```
Console.WriteLine(account.GetAccountHistory());
```

Run your program to see the results.

Next steps

If you got stuck, you can see the source for this tutorial [in our GitHub repo](#).

You can continue with the [object oriented programming](#) tutorial.

You can learn more about these concepts in these articles:

- [If and else statement](#)
- [While statement](#)
- [Do statement](#)
- [For statement](#)

Object-Oriented programming (C#)

3/25/2021 • 11 minutes to read • [Edit Online](#)

C# is an object-oriented language. Four of the key techniques used in object-oriented programming are:

- *Abstraction* means that a group of related properties, methods, and other members are treated as a single unit or object.
- *Encapsulation* means hiding the unnecessary details from type consumers.
- *Inheritance* describes the ability to create new classes based on an existing class.
- *Polymorphism* means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

In the preceding tutorial, [introduction to classes](#) you saw both *abstraction* and *encapsulation*. The `BankAccount` class provided an abstraction for the concept of a bank account. You could modify its implementation without affecting any of the code that used the `BankAccount` class. Both the `BankAccount` and `Transaction` classes provide encapsulation of the components needed to describe those concepts in code.

In this tutorial, you'll extend that application to make use of *inheritance* and *polymorphism* to add new features. You'll also add features to the `BankAccount` class, taking advantage of the *abstraction* and *encapsulation* techniques you learned in the preceding tutorial.

Create different types of accounts

After building this program, you get requests to add features to it. It works great in the situation where there is only one bank account type. Over time, needs change, and related account types are requested:

- An interest earning account that accrues interest at the end of each month.
- A line of credit that can have a negative balance, but when there's a balance, there's an interest charge each month.
- A pre-paid gift card account that starts with a single deposit, and only can be paid off. It can be refilled once at the start of each month.

All of these different accounts are similar to `BankAccount` class defined in the earlier tutorial. You could copy that code, rename the classes, and make modifications. That technique would work in the short term, but it would be more work over time. Any changes would be copied across all the affected classes.

Instead, you can create new bank account types that inherit methods and data from the `BankAccount` class created in the preceding tutorial. These new classes can extend the `BankAccount` class with the specific behavior needed for each type:

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Each of these classes *inherits* the shared behavior from their shared *base class*, the `BankAccount` class. Write the implementations for new and different functionality in each of the *derived classes*. These derived classes already have all the behavior defined in the `BankAccount` class.

It's a good practice to create each new class in a different source file. In [Visual Studio](#), you can right-click on the project, and select *add class* to add a new class in a new file. In [Visual Studio Code](#), select *File* then *New* to create a new source file. In either tool, name the file to match the class: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs*, and *GiftCardAccount.cs*.

When you create the classes as shown in the preceding sample, you'll find that none of your derived classes compile. A constructor is responsible for initializing an object. A derived class constructor must initialize the derived class, and provide instructions on how to initialize the base class object included in the derived class. The proper initialization normally happens without any extra code. The `BankAccount` class declares one public constructor with the following signature:

```
public BankAccount(string name, decimal initialBalance)
```

The compiler doesn't generate a default constructor when you define a constructor yourself. That means each derived class must explicitly call this constructor. You declare a constructor that can pass arguments to the base class constructor. The following code shows the constructor for the `InterestEarningAccount`:

```
public InterestEarningAccount(string name, decimal initialBalance) : base(name, initialBalance)
{
}
```

The parameters to this new constructor match the parameter type and names of the base class constructor. You use the `: base()` syntax to indicate a call to a base class constructor. Some classes define multiple constructors, and this syntax enables you to pick which base class constructor you call. Once you've updated the constructors, you can develop the code for each of the derived classes. The requirements for the new classes can be stated as follows:

- An interest earning account:
 - Will get a credit of 2% of the month-ending-balance.
- A line of credit:
 - Can have a negative balance, but not be greater in absolute value than the credit limit.
 - Will incur an interest charge each month where the end of month balance isn't 0.
 - Will incur a fee on each withdrawal that goes over the credit limit.
- A gift card account:
 - Can be refilled with a specified amount once each month, on the last day of the month.

You can see that all three of these account types have an action that takes places at the end of each month. However, each account type does different tasks. You use *polymorphism* to implement this code. Create a single `virtual` method in the `BankAccount` class:

```
public virtual void PerformMonthEndTransactions() { }
```

The preceding code shows how you use the `virtual` keyword to declare a method in the base class that a derived class may provide a different implementation for. A `virtual` method is a method where any derived class may choose to reimplement. The derived classes use the `override` keyword to define the new implementation. Typically you refer to this as "overriding the base class implementation". The `virtual` keyword specifies that derived classes may override the behavior. You can also declare `abstract` methods where derived

classes must override the behavior. The base class does not provide an implementation for an `abstract` method. Next, you need to define the implementation for two of the new classes you've created. Start with the `InterestEarningAccount`:

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        var interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Add the following code to the `LineOfCreditAccount`. The code negates the balance to compute a positive interest charge that is withdrawn from the account:

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        var interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

The `GiftCardAccount` class needs two changes to implement its month-end functionality. First, modify the constructor to include an optional amount to add each month:

```
private decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal monthlyDeposit = 0) : base(name,
initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

The constructor provides a default value for the `monthlyDeposit` value so callers can omit a `0` for no monthly deposit. Next, override the `PerformMonthEndTransactions` method to add the monthly deposit, if it was set to a non-zero value in the constructor:

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

The override applies the monthly deposit set in the constructor. Add the following code to the `Main` method to test these changes for the `GiftCardAccount` and the `InterestEarningAccount`:

```

var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 1000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());

```

Verify the results. Now, add a similar set of test code for the `LineOfCreditAccount` :

```

var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());

```

When you add the preceding code and run the program, you'll see something like the following error:

```

Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter 'amount')
   at OOPProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in
BankAccount.cs:line 42
   at OOPProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
   at OOPProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in
LineOfCreditAccount.cs:line 9
   at OOPProgramming.Program.Main(String[] args) in Program.cs:line 29

```

NOTE

The actual output includes the full path to the folder with the project. The folder names were omitted for brevity. Also, depending on your code format, the line numbers may be slightly different.

This code fails because the `BankAccount` assumes that the initial balance must be greater than 0. Another assumption baked into the `BankAccount` class is that the balance can't go negative. Instead, any withdrawal that overdraws the account is rejected. Both of those assumptions need to change. The line of credit account starts at 0, and generally will have a negative balance. Also, if a customer borrows too much money, they incur a fee. The transaction is accepted, it just costs more. The first rule can be implemented by adding an optional argument to the `BankAccount` constructor that specifies the minimum balance. The default is `0`. The second rule requires a mechanism that enables derived classes to modify the default algorithm. In a sense, the base class "asks" the derived type what should happen when there's an overdraft. The default behavior is to reject the transaction by throwing an exception.

Let's start by adding a second constructor that includes an optional `minimumBalance` parameter. This new constructor does all the actions done by the existing constructor. Also, it sets the `minimumBalance` property. You could copy the body of the existing constructor, but that means two locations to change in the future. Instead, you can use *constructor chaining* to have one constructor call another. The following code shows the two

constructors and the new additional field:

```
private readonly decimal minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name, initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    this.minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

The preceding code shows two new techniques. First, the `minimumBalance` field is marked as `readonly`. That means the value cannot be changed after the object is constructed. Once a `BankAccount` is created, the `minimumBalance` can't change. Second, the constructor that takes two parameters uses `: this(name, initialBalance, 0) { }` as its implementation. The `: this()` expression calls the other constructor, the one with three parameters. This technique allows you to have a single implementation for initializing an object even though client code can choose one of many constructors.

This implementation calls `MakeDeposit` only if the initial balance is greater than `0`. That preserves the rule that deposits must be positive, yet lets the credit account open with a `0` balance.

Now that the `BankAccount` class has a read-only field for the minimum balance, the final change is to change the hard code `0` to `minimumBalance` in the `MakeWithdrawal` method:

```
if (Balance - amount < minimumBalance)
```

After extending the `BankAccount` class, you can modify the `LineOfCreditAccount` constructor to call the new base constructor, as shown in the following code:

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name,
initialBalance, -creditLimit)
{ }
```

Notice that the `LineOfCreditAccount` constructor changes the sign of the `creditLimit` parameter so it matches the meaning of the `minimumBalance` parameter.

Different overdraft rules

The last feature to add enables the `LineOfCreditAccount` to charge a fee for going over the credit limit instead of refusing the transaction.

One technique is to define a virtual function where you implement the required behavior. The `BankAccount` class refactors the `MakeWithdrawal` method into two methods. The new method does the specified action when the withdrawal takes the balance below the minimum. The existing `MakeWithdrawal` method has the following code:

```

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}

```

Replace it with the following code:

```

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    var overdraftTransaction = CheckWithdrawalLimit(Balance - amount < minimumBalance);
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    else
    {
        return default;
    }
}

```

The added method is `protected`, which means that it can be called only from derived classes. That declaration prevents other clients from calling the method. It's also `virtual` so that derived classes can change the behavior. The return type is a `Transaction?`. The `?` annotation indicates that the method may return `null`. Add the following implementation in the `LineOfCreditAccount` to charge a fee when the withdrawal limit is exceeded:

```

protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;

```

The override returns a fee transaction when the account is overdrawn. If the withdrawal doesn't go over the limit, the method returns a `null` transaction. That indicates there's no fee. Test these changes by adding the following code to your `Main` method in the `Program` class:

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Run the program, and check the results.

Summary

If you got stuck, you can see the source for this tutorial [in our GitHub repo](#).

This tutorial demonstrated many of the techniques used in Object-Oriented programming:

- You used *Abstraction* when you defined classes for each of the different account types. Those classes described the behavior for that type of account.
- You used *Encapsulation* when you kept many details `private` in each class.
- You used *Inheritance* when you leveraged the implementation already created in the `BankAccount` class to save code.
- You used *Polymorphism* when you created `virtual` methods that derived classes could override to create specific behavior for that account type.

Congratulations, you've finished all of our introduction to C# tutorials. To learn more, try more of our [tutorials](#).

Use string interpolation to construct formatted strings

3/6/2021 • 7 minutes to read • [Edit Online](#)

This tutorial teaches you how to use C# [string interpolation](#) to insert values into a single result string. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that show you how to insert values into a string and format those values in different ways.

This tutorial expects that you have a machine you can use for development. The .NET tutorial [Hello World in 10 minutes](#) has instructions for setting up your local development environment on Windows, Linux, or macOS. You can also complete the [interactive version](#) of this tutorial in your browser.

Create an interpolated string

Create a directory named *interpolated*. Make it the current directory and run the following command from a console window:

```
dotnet new console
```

This command creates a new .NET Core console application in the current directory.

Open *Program.cs* in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following code, where you replace `<name>` with your name:

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Try this code by typing `dotnet run` in your console window. When you run the program, it displays a single string that includes your name in the greeting. The string included in the [WriteLine](#) method call is an *interpolated string expression*. It's a kind of template that lets you construct a single string (called the *result string*) from a string that includes embedded code. Interpolated strings are particularly useful for inserting values into a string or concatenating (joining together) strings.

This simple example contains the two elements that every interpolated string must have:

- A string literal that begins with the `$` character before its opening quotation mark character. There can't be any spaces between the `$` symbol and the quotation mark character. (If you'd like to see what happens if you include one, insert a space after the `$` character, save the file, and run the program again by typing `dotnet run` in the console window. The C# compiler displays an error message, "error CS1056: Unexpected character '\$'".)
- One or more *interpolation expressions*. An interpolation expression is indicated by an opening and closing brace (`{` and `}`). You can put any C# expression that returns a value (including `null`) inside the braces.

Let's try a few more string interpolation examples with some other data types.

Include different data types

In the previous section, you used string interpolation to insert one string inside of another. The result of an

interpolation expression can be of any data type, though. Let's include values of various data types in an interpolated string.

In the following example, we first define a `class` data type `Vegetable` that has a `Name` `property` and a `ToString` `method`, which `overrides` the behavior of the `Object.ToString()` method. The `public` `access modifier` makes that method available to any client code to get the string representation of a `Vegetable` instance. In the example the `Vegetable.ToString` method returns the value of the `Name` `property` that is initialized at the `Vegetable` `constructor`:

```
public Vegetable(string name) => Name = name;
```

Then we create an instance of the `Vegetable` class named `item` by using the `new` `operator` and providing a name for the constructor `Vegetable`:

```
var item = new Vegetable("eggplant");
```

Finally, we include the `item` variable into an interpolated string that also contains a `DateTime` value, a `Decimal` value, and a `Unit` `enumeration` value. Replace all of the C# code in your editor with the following code, and then use the `dotnet run` command to run it:

```
using System;

public class Vegetable
{
    public Vegetable(string name) => Name = name;

    public string Name { get; }

    public override string ToString() => Name;
}

public class Program
{
    public enum Unit { item, kilogram, gram, dozen };

    public static void Main()
    {
        var item = new Vegetable("eggplant");
        var date = DateTime.Now;
        var price = 1.99m;
        var unit = Unit.item;
        Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
    }
}
```

Note that the interpolation expression `item` in the interpolated string resolves to the text "eggplant" in the result string. That's because, when the type of the expression result is not a string, the result is resolved to a string in the following way:

- If the interpolation expression evaluates to `null`, an empty string ("", or `String.Empty`) is used.
- If the interpolation expression doesn't evaluate to `null`, typically the `ToString` method of the result type is called. You can test this by updating the implementation of the `Vegetable.ToString` method. You might not even need to implement the `ToString` method since every type has some implementation of this method. To test this, comment out the definition of the `Vegetable.ToString` method in the example (to do that, put a comment symbol, `//`, in front of it). In the output, the string "eggplant" is replaced by the fully qualified type name ("Vegetable" in this example), which is the default behavior of the `Object.ToString()`

method. The default behavior of the `Tostring` method for an enumeration value is to return the string representation of the value.

In the output from this example, the date is too precise (the price of eggplant doesn't change every second), and the price value doesn't indicate a unit of currency. In the next section, you'll learn how to fix those issues by controlling the format of string representations of the expression results.

Control the formatting of interpolation expressions

In the previous section, two poorly formatted strings were inserted into the result string. One was a date and time value for which only the date was appropriate. The second was a price that didn't indicate its unit of currency. Both issues are easy to address. String interpolation lets you specify *format strings* that control the formatting of particular types. Modify the call to `Console.WriteLine` from the previous example to include the format strings for the date and price expressions as shown in the following line:

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

You specify a format string by following the interpolation expression with a colon (":") and the format string. "d" is a [standard date and time format string](#) that represents the short date format. "C2" is a [standard numeric format string](#) that represents a number as a currency value with two digits after the decimal point.

A number of types in the .NET libraries support a predefined set of format strings. These include all the numeric types and the date and time types. For a complete list of types that support format strings, see [Format Strings and .NET Class Library Types](#) in the [Formatting Types in .NET](#) article.

Try modifying the format strings in your text editor and, each time you make a change, rerun the program to see how the changes affect the formatting of the date and time and the numeric value. Change the "d" in `{date:d}` to "t" (to display the short time format), "y" (to display the year and month), and "yyyy" (to display the year as a four-digit number). Change the "C2" in `{price:C2}` to "e" (for exponential notation) and "F3" (for a numeric value with three digits after the decimal point).

In addition to controlling formatting, you can also control the field width and alignment of the formatted strings that are included in the result string. In the next section, you'll learn how to do this.

Control the field width and alignment of interpolation expressions

Ordinarily, when the result of an interpolation expression is formatted to string, that string is included in a result string without leading or trailing spaces. Particularly when you work with a set of data, being able to control a field width and text alignment helps to produce a more readable output. To see this, replace all the code in your text editor with the following code, then type `dotnet run` to execute the program:

```

using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var titles = new Dictionary<string, string>()
        {
            ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
            ["London, Jack"] = "Call of the Wild, The",
            ["Shakespeare, William"] = "Tempest, The"
        };

        Console.WriteLine("Author and Title List");
        Console.WriteLine();
        Console.WriteLine($"|{\"Author\", -25}|{\"Title\", 30}|");
        foreach (var title in titles)
            Console.WriteLine($"|{title.Key, -25}|{title.Value, 30}|");
    }
}

```

The names of authors are left-aligned, and the titles they wrote are right-aligned. You specify the alignment by adding a comma (",") after an interpolation expression and designating the *minimum* field width. If the specified value is a positive number, the field is right-aligned. If it is a negative number, the field is left-aligned.

Try removing the negative signs from the `{"Author", -25}` and `{title.Key, -25}` code and run the example again, as the following code does:

```

Console.WriteLine($"|{\"Author\", 25}|{\"Title\", 30}|");
foreach (var title in titles)
    Console.WriteLine($"|{title.Key, 25}|{title.Value, 30}|");

```

This time, the author information is right-aligned.

You can combine an alignment specifier and a format string for a single interpolation expression. To do that, specify the alignment first, followed by a colon and the format string. Replace all of the code inside the `Main` method with the following code, which displays three formatted strings with defined field widths. Then run the program by entering the `dotnet run` command.

```
Console.WriteLine($"{DateTime.Now, -20:d} {DateTime.Now, -10:HH} [{1063.342, 15:N2}] feet");
```

The output looks something like the following:

[04/14/2018] Hour [16] [1,063.34] feet
-------------	------------	-----	----------------

You've completed the string interpolation tutorial.

For more information, see the [String interpolation](#) topic and the [String interpolation in C#](#) tutorial.

String interpolation in C#

10/29/2019 • 5 minutes to read • [Edit Online](#)

This tutorial shows you how to use [string interpolation](#) to format and include expression results in a result string. The examples assume that you are familiar with basic C# concepts and .NET type formatting. If you are new to string interpolation or .NET type formatting, check out the [interactive string interpolation tutorial](#) first. For more information about formatting types in .NET, see the [Formatting Types in .NET](#) topic.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

Introduction

The [string interpolation](#) feature is built on top of the [composite formatting](#) feature and provides a more readable and convenient syntax to include formatted expression results in a result string.

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You can embed any valid C# expression that returns a value in an interpolated string. In the following example, as soon as an expression is evaluated, its result is converted into a string and included in a result string:

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is
{CalculateHypotenuse(a, b)}");

double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);

// Expected output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

As the example shows, you include an expression in an interpolated string by enclosing it with braces:

```
{<interpolationExpression>}
```

Interpolated strings support all the capabilities of the [string composite formatting](#) feature. That makes them a more readable alternative to the use of the [String.Format](#) method.

How to specify a format string for an interpolation expression

You specify a format string that is supported by the type of the expression result by following the interpolation expression with a colon (":") and the format string:

```
{<interpolationExpression>:<formatString>}
```

The following example shows how to specify standard and custom format strings for expressions that produce date and time or numeric results:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

For more information, see the [Format String Component](#) section of the [Composite Formatting](#) topic. That section provides links to the topics that describe standard and custom format strings supported by .NET base types.

How to control the field width and alignment of the formatted interpolation expression

You specify the minimum field width and the alignment of the formatted expression result by following the interpolation expression with a comma (",") and the constant expression:

```
{<interpolationExpression>,<alignment>}
```

If the *alignment* value is positive, the formatted expression result is right-aligned; if negative, it's left-aligned.

If you need to specify both alignment and a format string, start with the alignment component:

```
{<interpolationExpression>,<alignment>:<formatString>}
```

The following example shows how to specify alignment and uses pipe characters ("|") to delimit text fields:

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a + b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a * b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic| 3.500|
// |Geometric| 3.464|
// |Harmonic | 3.429|
```

As the example output shows, if the length of the formatted expression result exceeds specified field width, the *alignment* value is ignored.

For more information, see the [Alignment Component](#) section of the [Composite Formatting](#) topic.

How to use escape sequences in an interpolated string

Interpolated strings support all escape sequences that can be used in ordinary string literals. For more information, see [String escape sequences](#).

To interpret escape sequences literally, use a [verbatim](#) string literal. An interpolated verbatim string starts with the `$` character followed by the `@` character. Starting with C# 8.0, you can use the `$` and `@` tokens in any order: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings.

To include a brace, "{" or "}", in a result string, use two braces, "{{{" or "}}". For more information, see the [Escaping Braces](#) section of the [Composite Formatting](#) topic.

The following example shows how to include braces in a result string and construct a verbatim interpolated string:

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and {{{string.Join(", ",ys)}}} sets.");
Console.WriteLine(string.Join(", ", xs));
Console.WriteLine(string.Join(", ", ys));

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\\Users\\{userName}\\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\\Users\\Jane\\Documents
// C:\\Users\\Jane\\Documents
```

How to use a ternary conditional operator `? :` in an interpolation expression

As the colon (":") has special meaning in an item with an interpolation expression, in order to use a [conditional operator](#) in an expression, enclose it in parentheses, as the following example shows:

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {((rand.NextDouble() < 0.5) ? "heads" : "tails")}");
}
```

How to create a culture-specific result string with string interpolation

By default, an interpolated string uses the current culture defined by the [CultureInfo.CurrentCulture](#) property for all formatting operations. Use implicit conversion of an interpolated string to a [System.FormattableString](#) instance and call its [ToString\(IFormatProvider\)](#) method to create a culture-specific result string. The following example shows how to do that:

```

var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};

var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US      5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL      17-05-18 15:44:55      31.415.926,536
//          05/17/2018 15:44:55      31,415,926.536

```

As the example shows, you can use one [FormattableString](#) instance to generate multiple result strings for various cultures.

How to create a result string using the invariant culture

Along with the [FormattableString.ToString\(IFormatProvider\)](#) method, you can use the static [FormattableString.Invariant](#) method to resolve an interpolated string to a result string for the [InvariantCulture](#). The following example shows how to do that:

```

string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24

```

Conclusion

This tutorial describes common scenarios of string interpolation usage. For more information about string interpolation, see the [String interpolation](#) topic. For more information about formatting types in .NET, see the [Formatting Types in .NET](#) and [Composite formatting](#) topics.

See also

- [String.Format](#)
- [System.FormattableString](#)
- [System.IFormattable](#)
- [Strings](#)

Console app

3/23/2021 • 12 minutes to read • [Edit Online](#)

This tutorial teaches you a number of features in .NET Core and the C# language. You'll learn:

- The basics of the .NET Core CLI
- The structure of a C# Console Application
- Console I/O
- The basics of File I/O APIs in .NET
- The basics of the Task-based Asynchronous Programming in .NET

You'll build an application that reads a text file, and echoes the contents of that text file to the console. The output to the console is paced to match reading it aloud. You can speed up or slow down the pace by pressing the '<' (less than) or '>' (greater than) keys.

There are a lot of features in this tutorial. Let's build them one by one.

Prerequisites

- Set up your machine to run .NET Core. You can find the installation instructions on the [.NET Core downloads](#) page. You can run this application on Windows, Linux, macOS, or in a Docker container.
- Install your favorite code editor.

Create the app

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

Before you start making modifications, let's go through the steps to run the simple Hello World application. After creating the application, type `dotnet restore` at the command prompt. This command runs the NuGet package restore process. NuGet is a .NET package manager. This command downloads any of the missing dependencies for your project. As this is a new project, none of the dependencies are in place, so the first run will download the .NET Core framework. After this initial step, you will only need to run `dotnet restore` when you add new dependent packages, or update the versions of any of your dependencies.

You don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore documentation](#).

After restoring packages, you run `dotnet build`. This executes the build engine and creates your application executable. Finally, you execute `dotnet run` to run your application.

The simple Hello World application code is all in Program.cs. Open that file with your favorite text editor. We're about to make our first changes. At the top of the file, see a using statement:

```
using System;
```

This statement tells the compiler that any types from the `System` namespace are in scope. Like other Object Oriented languages you may have used, C# uses namespaces to organize types. This Hello World program is no different. You can see that the program is enclosed in the namespace with the name based on the name of the current directory. For this tutorial, let's change the name of the namespace to `TeleprompterConsole`:

```
namespace TeleprompterConsole
```

Reading and Echoing the File

The first feature to add is the ability to read a text file and display all that text to the console. First, let's add a text file. Copy the `sampleQuotes.txt` file from the GitHub repository for this `sample` into your project directory. This will serve as the script for your application. If you would like information on how to download the sample app for this topic, see the instructions in the [Samples and Tutorials](#) topic.

Next, add the following method in your `Program` class (right below the `Main` method):

```
static IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

This method uses types from two new namespaces. For this to compile you'll need to add the following two lines to the top of the file:

```
using System.Collections.Generic;
using System.IO;
```

The `IEnumerable<T>` interface is defined in the `System.Collections.Generic` namespace. The `File` class is defined in the `System.IO` namespace.

This method is a special type of C# method called an *Iterator method*. Enumerator methods return sequences that are evaluated lazily. That means each item in the sequence is generated as it is requested by the code consuming the sequence. Enumerator methods are methods that contain one or more `yield return` statements. The object returned by the `ReadFrom` method contains the code to generate each item in the sequence. In this example, that involves reading the next line of text from the source file, and returning that string. Each time the calling code requests the next item from the sequence, the code reads the next line of text from the file and returns it. When the file is completely read, the sequence indicates that there are no more items.

There are two other C# syntax elements that may be new to you. The `using` statement in this method manages resource cleanup. The variable that is initialized in the `using` statement (`reader`, in this example) must implement the `IDisposable` interface. That interface defines a single method, `Dispose`, that should be called when the resource should be released. The compiler generates that call when execution reaches the closing brace of the `using` statement. The compiler-generated code ensures that the resource is released even if an

exception is thrown from the code in the block defined by the using statement.

The `reader` variable is defined using the `var` keyword. `var` defines an *implicitly typed local variable*. That means the type of the variable is determined by the compile-time type of the object assigned to the variable. Here, that is the return value from the `OpenText(String)` method, which is a `StreamReader` object.

Now, let's fill in the code to read the file in the `Main` method:

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Run the program (using `dotnet run`) and you can see every line printed out to the console.

Adding Delays and Formatting output

What you have is being displayed far too fast to read aloud. Now you need to add the delays in the output. As you start, you'll be building some of the core code that enables asynchronous processing. However, these first steps will follow a few anti-patterns. The anti-patterns are pointed out in comments as you add the code, and the code will be updated in later steps.

There are two steps to this section. First, you'll update the iterator method to return single words instead of entire lines. That's done with these modifications. Replace the `yield return line;` statement with the following code:

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Next, you need to modify how you consume the lines of the file, and add a delay after writing each word.

Replace the `Console.WriteLine(line)` statement in the `Main` method with the following block:

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

The `Task` class is in the `System.Threading.Tasks` namespace, so you need to add that `using` statement at the top of file:

```
using System.Threading.Tasks;
```

Run the sample, and check the output. Now, each single word is printed, followed by a 200 ms delay. However, the displayed output shows some issues because the source text file has several lines that have more than 80 characters without a line break. That can be hard to read while it's scrolling by. That's easy to fix. You'll just keep track of the length of each line, and generate a new line whenever the line length reaches a certain threshold.

Declare a local variable after the declaration of `words` in the `ReadFrom` method that holds the line length:

```
var lineLength = 0;
```

Then, add the following code after the `yield return word + " "` statement (before the closing brace):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Run the sample, and you'll be able to read aloud at its pre-configured pace.

Async Tasks

In this final step, you'll add the code to write the output asynchronously in one task, while also running another task to read input from the user if they want to speed up or slow down the text display, or stop the text display altogether. This has a few steps in it and by the end, you'll have all the updates that you need. The first step is to create an asynchronous `Task` returning method that represents the code you've created so far to read and display the file.

Add this method to your `Program` class (it's taken from the body of your `Main` method):

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

You'll notice two changes. First, in the body of the method, instead of calling `Wait()` to synchronously wait for a task to finish, this version uses the `await` keyword. In order to do that, you need to add the `async` modifier to the method signature. This method returns a `Task`. Notice that there are no return statements that return a `Task` object. Instead, that `Task` object is created by code the compiler generates when you use the `await` operator. You can imagine that this method returns when it reaches an `await`. The returned `Task` indicates that the work has not completed. The method resumes when the awaited task completes. When it has executed to completion, the returned `Task` indicates that it is complete. Calling code can monitor that returned `Task` to determine when it has completed.

You can call this new method in your `Main` method:

```
ShowTeleprompter().Wait();
```

Here, in `Main`, the code does synchronously wait. You should use the `await` operator instead of synchronously waiting whenever possible. But, in a console application's `Main` method, you cannot use the `await` operator. That would result in the application exiting before all tasks have completed.

NOTE

If you use C# 7.1 or later, you can create console applications with `async` `Main` method.

Next, you need to write the second asynchronous method to read from the Console and watch for the '<' (less than), '>' (greater than) and 'X' or 'x' keys. Here's the method you add for that task:

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

This creates a lambda expression to represent an `Action` delegate that reads a key from the Console and modifies a local variable representing the delay when the user presses the '<' (less than) or '>' (greater than) keys. The delegate method finishes when user presses the 'X' or 'x' keys, which allow the user to stop the text display at any time. This method uses `ReadKey()` to block and wait for the user to press a key.

To finish this feature, you need to create a new `async Task` returning method that starts both of these tasks (`GetInput` and `ShowTeleprompter`), and also manages the shared data between these two tasks.

It's time to create a class that can handle the shared data between these two tasks. This class contains two public properties: the delay, and a flag `Done` to indicate that the file has been completely read:

```

namespace TeleprompterConsole
{
    internal class TelePrompterConfig
    {
        public int DelayInMilliseconds { get; private set; } = 200;

        public void UpdateDelay(int increment) // negative to speed up
        {
            var newDelay = Min(DelayInMilliseconds + increment, 1000);
            newDelay = Max(newDelay, 20);
            DelayInMilliseconds = newDelay;
        }

        public bool Done { get; private set; }

        public void SetDone()
        {
            Done = true;
        }
    }
}

```

Put that class in a new file, and enclose that class in the `TeleprompterConsole` namespace as shown above. You'll also need to add a `using static` statement so that you can reference the `Min` and `Max` methods without the enclosing class or namespace names. A `using static` statement imports the methods from one class. This is in contrast with the `using` statements used up to this point that have imported all classes from a namespace.

```
using static System.Math;
```

Next, you need to update the `ShowTeleprompter` and `GetInput` methods to use the new `config` object. Write one final `Task` returning `async` method to start both tasks and exit when the first task finishes:

```

private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}

```

The one new method here is the `WhenAny(Task[])` call. That creates a `Task` that finishes as soon as any of the tasks in its argument list completes.

Next, you need to update both the `ShowTeleprompter` and `GetInput` methods to use the `config` object for the delay:

```

private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}

```

This new version of `ShowTeleprompter` calls a new method in the `TeleprompterConfig` class. Now, you need to update `Main` to call `RunTeleprompter` instead of `ShowTeleprompter`:

```
RunTeleprompter().Wait();
```

Conclusion

This tutorial showed you a number of the features around the C# language and the .NET Core libraries related to working in Console applications. You can build on this knowledge to explore more about the language, and the classes introduced here. You've seen the basics of File and Console I/O, blocking and non-blocking use of the Task-based asynchronous programming, a tour of the C# language and how C# programs are organized, and the .NET Core CLI.

For more information about File I/O, see the [File and Stream I/O](#) topic. For more information about asynchronous programming model used in this tutorial, see the [Task-based Asynchronous Programming](#) topic and the [Asynchronous programming](#) topic.

REST client

3/23/2021 • 11 minutes to read • [Edit Online](#)

This tutorial teaches you a number of features in .NET Core and the C# language. You'll learn:

- The basics of the .NET Core CLI.
- An overview of C# Language features.
- Managing dependencies with NuGet
- HTTP Communications
- Processing JSON information
- Managing configuration with Attributes.

You'll build an application that issues HTTP Requests to a REST service on GitHub. You'll read information in JSON format, and convert that JSON packet into C# objects. Finally, you'll see how to work with C# objects.

There are many features in this tutorial. Let's build them one by one.

If you prefer to follow along with the [final sample](#) for this article, you can download it. For download instructions, see [Samples and Tutorials](#).

Prerequisites

You'll need to set up your machine to run .NET core. You can find the installation instructions on the [.NET Core Downloads](#) page. You can run this application on Windows, Linux, or macOS, or in a Docker container. You'll need to install your favorite code editor. The descriptions below use [Visual Studio Code](#), which is an open source, cross platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Enter the following command in a console window:

```
dotnet new console --name WebAPIClient
```

This creates the starter files for a basic "Hello World" application. The project name is "WebAPIClient". As this is a new project, none of the dependencies are in place. The first run will download the .NET Core framework, install a development certificate, and run the NuGet package manager to restore missing dependencies.

Before you start making modifications, `cd` into the "WebAPIClient" directory and type `dotnet run` ([see note](#)) at the command prompt to run your application. `dotnet run` automatically performs `dotnet restore` if your environment is missing dependencies. It also performs `dotnet build` if your application needs to be rebuilt. After your initial setup, you will only need to run `dotnet restore` or `dotnet build` when it makes sense for your project.

Adding New Dependencies

One of the key design goals for .NET Core is to minimize the size of the .NET installation. If an application needs additional libraries for some of its features, you add those dependencies into your C# project (*.csproj) file. For our example, you'll need to add the `System.Runtime.Serialization.Json` package, so your application can process JSON responses.

You'll need the `System.Runtime.Serialization.Json` package for this application. Add it to your project by running the following .NET CLI command:

```
dotnet add package System.Text.Json
```

Making Web Requests

Now you're ready to start retrieving data from the web. In this application, you'll read information from the [GitHub API](#). Let's read information about the projects under the [.NET Foundation](#) umbrella. You'll start by making the request to the GitHub API to retrieve information on the projects. The endpoint you'll use is: <https://api.github.com/orgs/dotnet/repos>. You want to retrieve all the information about these projects, so you'll use an HTTP GET request. Your browser also uses HTTP GET requests, so you can paste that URL into your browser to see what information you'll be receiving and processing.

You use the [HttpClient](#) class to make web requests. Like all modern .NET APIs, [HttpClient](#) supports only async methods for its long-running APIs. Start by making an async method. You'll fill in the implementation as you build the functionality of the application. Start by opening the `program.cs` file in your project directory and adding the following method to the `Program` class:

```
private static async Task ProcessRepositories()  
{  
}
```

You'll need to add a `using` directive at the top of your `Main` method so that the C# compiler recognizes the [Task](#) type:

```
using System.Threading.Tasks;
```

If you build your project at this point, you'll get a warning generated for this method, because it does not contain any `await` operators and will run synchronously. Ignore that for now; you'll add `await` operators as you fill in the method.

Next, update the `Main` method to call the `ProcessRepositories` method. The `ProcessRepositories` method returns a task, and you shouldn't exit the program before that task finishes. Therefore, you must change the signature of `Main`. Add the `async` modifier, and change the return type to `Task`. Then, in the body of the method, add a call to `ProcessRepositories`. Add the `await` keyword to that method call:

```
static async Task Main(string[] args)  
{  
    await ProcessRepositories();  
}
```

Now, you have a program that does nothing, but does it asynchronously. Let's improve it.

First you need an object that is capable to retrieve data from the web; you can use a [HttpClient](#) to do that. This object handles the request and the responses. Instantiate a single instance of that type in the `Program` class inside the `Program.cs` file.

```

namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}

```

Let's go back to the `ProcessRepositories` method and fill in a first version of it:

```

private static async Task ProcessRepositories()
{
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
    client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository Reporter");

    var stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.WriteLine(msg);
}

```

You'll need to also add two new `using` directives at the top of the file for this to compile:

```

using System.Net.Http;
using System.Net.Http.Headers;

```

This first version makes a web request to read the list of all repositories under the dotnet foundation organization. (The GitHub ID for the .NET Foundation is `dotnet`.) The first few lines set up the `HttpClient` for this request. First, it is configured to accept the GitHub JSON responses. This format is simply JSON. The next line adds a User Agent header to all requests from this object. These two headers are checked by the GitHub server code, and are necessary to retrieve information from GitHub.

After you've configured the `HttpClient`, you make a web request and retrieve the response. In this first version, you use the `HttpClient.GetStringAsync(String)` convenience method. This convenience method starts a task that makes the web request, and then when the request returns, it reads the response stream and extracts the content from the stream. The body of the response is returned as a `String`. The string is available when the task completes.

The final two lines of this method await that task, and then print the response to the console. Build the app, and run it. The build warning is gone now, because the `ProcessRepositories` now does contain an `await` operator. You'll see a long display of JSON formatted text.

Processing the JSON Result

At this point, you've written the code to retrieve a response from a web server, and display the text that is contained in that response. Next, let's convert that JSON response into C# objects.

The `System.Text.Json.JsonSerializer` class serializes objects to JSON and deserializes JSON into objects. Start by defining a class to represent the `repo` JSON object returned from the GitHub API:

```

using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}

```

Put the above code in a new file called 'repo.cs'. This version of the class represents the simplest path to process JSON data. The class name and the member name match the names used in the JSON packet, instead of following C# conventions. You'll fix that by providing some configuration attributes later. This class demonstrates another important feature of JSON serialization and deserialization: Not all the fields in the JSON packet are part of this class. The JSON serializer will ignore information that is not included in the class type being used. This feature makes it easier to create types that work with only a subset of the fields in the JSON packet.

Now that you've created the type, let's deserialize it.

Next, you'll use the serializer to convert JSON into C# objects. Replace the call to [GetStringAsync\(String\)](#) in your `ProcessRepositories` method with the following lines:

```

var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);

```

You're using new namespaces, so you'll need to add it at the top of the file as well:

```

using System.Collections.Generic;
using System.Text.Json;

```

Notice that you're now using [GetStreamAsync\(String\)](#) instead of [GetStringAsync\(String\)](#). The serializer uses a stream instead of a string as its source. Let's explain a couple features of the C# language that are being used in the second line of the preceding code snippet. The first argument to [JsonSerializer.DeserializeAsync< TValue > \(Stream, JsonSerializerOptions, CancellationToken\)](#) is an `await` expression. (The other two parameters are optional and are omitted in the code snippet.) Await expressions can appear almost anywhere in your code, even though up to now, you've only seen them as part of an assignment statement. The `Deserialize` method is *generic*, which means you must supply type arguments for what kind of objects should be created from the JSON text. In this example, you're deserializing to a `List<Repository>`, which is another generic object, the `System.Collections.Generic.List< T >`. The `List< >` class stores a collection of objects. The type argument declares the type of objects stored in the `List< >`. The JSON text represents a collection of repo objects, so the type argument is `Repository`.

You're almost done with this section. Now that you've converted the JSON to C# objects, let's display the name of each repository. Replace the lines that read:

```

var msg = await stringTask;    /**Deleted this
Console.WriteLine(msg);

```

with the following:

```

foreach (var repo in repositories)
    Console.WriteLine(repo.name);

```

Compile and run the application. It will print the names of the repositories that are part of the .NET Foundation.

Controlling Serialization

Before you add more features, let's address the `name` property by using the `[JsonPropertyName]` attribute. Make the following changes to the declaration of the `name` field in `repo.cs`:

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

To use `[JsonPropertyName]` attribute, you will need to add the [System.Text.Json.Serialization](#) namespace to the `using` directives:

```
using System.Text.Json.Serialization;
```

This change means you need to change the code that writes the name of each repository in `program.cs`:

```
Console.WriteLine(repo.Name);
```

Execute `dotnet run` to make sure you've got the mappings correct. You should see the same output as before.

Let's make one more change before adding new features. The `ProcessRepositories` method can do the async work and return a collection of the repositories. Let's return the `List<Repository>` from that method, and move the code that writes the information into the `Main` method.

Change the signature of `ProcessRepositories` to return a task whose result is a list of `Repository` objects:

```
private static async Task<List<Repository>> ProcessRepositories()
```

Then, just return the repositories after processing the JSON response:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

The compiler generates the `Task<T>` object for the return because you've marked this method as `async`. Then, let's modify the `Main` method so that it captures those results and writes each repository name to the console. Your `Main` method now looks like this:

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

Reading More Information

Let's finish this by processing a few more of the properties in the JSON packet that gets sent from the GitHub API. You won't want to grab everything, but adding a few properties will demonstrate a few more features of the C# language.

Let's start by adding a few more simple types to the `Repository` class definition. Add these properties to that class:

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

These properties have built-in conversions from the string type (which is what the JSON packets contain) to the target type. The `Uri` type may be new to you. It represents a URI, or in this case, a URL. In the case of the `Uri` and `int` types, if the JSON packet contains data that does not convert to the target type, the serialization action will throw an exception.

Once you've added these, update the `Main` method to display those elements:

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

As a final step, let's add the information for the last push operation. This information is formatted in this fashion in the JSON response:

```
2016-02-08T21:27:00Z
```

That format is in Coordinated Universal Time (UTC) so you'll get a `DateTime` value whose `Kind` property is `Utc`. If you prefer a date represented in your time zone, you'll need to write a custom conversion method. First, define a `public` property that will hold the UTC representation of the date and time in your `Repository` class and a `LastPush` `readonly` property that returns the date converted to local time:

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

Let's go over the new constructs we just defined. The `LastPush` property is defined using an *expression-bodied member* for the `get` accessor. There is no `set` accessor. Omitting the `set` accessor is how you define a *read-only* property in C#. (Yes, you can create *write-only* properties in C#, but their value is limited.)

Finally, add one more output statement in the console, and you're ready to build and run this app again:

```
Console.WriteLine(repo.LastPush);
```

Your version should now match the [finished sample](#).

Conclusion

This tutorial showed you how to make web requests, parse the result, and display properties of those results. You've also added new packages as dependencies in your project. You've seen some of the features of the C# language that support object-oriented techniques.

You don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore documentation](#).

Inheritance in C# and .NET

11/2/2020 • 26 minutes to read • [Edit Online](#)

This tutorial introduces you to inheritance in C#. Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.

Prerequisites

This tutorial assumes that you've installed the .NET Core SDK. Visit the [.NET Core Downloads](#) page to download it. You also need a code editor. This tutorial uses [Visual Studio Code](#), although you can use any code editor of your choice.

Running the examples

To create and run the examples in this tutorial, you use the `dotnet` utility from the command line. Follow these steps for each example:

1. Create a directory to store the example.
2. Enter the `dotnet new console` command at a command prompt to create a new .NET Core project.
3. Copy and paste the code from the example into your code editor.
4. Enter the `dotnet restore` command from the command line to load or restore the project's dependencies.

You don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore](#) documentation.

5. Enter the `dotnet run` command to compile and execute the example.

Background: What is inheritance?

Inheritance is one of the fundamental attributes of object-oriented programming. It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class. The class whose members are inherited is called the *base class*. The class that inherits the members of the base class is called the *derived class*.

C# and .NET support *single inheritance* only. That is, a class can only inherit from a single class. However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types. In other words, type `D` can inherit from type `C`, which inherits from type `B`, which inherits from the base class type `A`. Because inheritance is transitive, the members of type `A` are available to type `D`.

Not all members of a base class are inherited by derived classes. The following members are not inherited:

- [Static constructors](#), which initialize the static data of a class.
- [Instance constructors](#), which you call to create a new instance of the class. Each class must define its own

constructors.

- **Finalizers**, which are called by the runtime's garbage collector to destroy instances of a class.

While all other members of a base class are inherited by derived classes, whether they are visible or not depends on their accessibility. A member's accessibility affects its visibility for derived classes as follows:

- **Private** members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes. In the following example, `A.B` is a nested class that derives from `A`, and `C` derives from `A`. The private `A.value` field is visible in `A.B`. However, if you remove the comments from the `C.GetValue` method and attempt to compile the example, it produces compiler error CS0122: "'A.value' is inaccessible due to its protection level."

```
using System;

public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
//    public int GetValue()
//    {
//        return this.value;
//    }
}

public class Example
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//      10
```

- **Protected** members are visible only in derived classes.
- **Internal** members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.
- **Public** members are visible in derived classes and are part of the derived class' public interface. Public inherited members can be called just as if they are defined in the derived class. In the following example, class `A` defines a method named `Method1`, and class `B` inherits from class `A`. The example then calls `Method1` as if it were an instance method on `B`.

```

public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        b.Method1();
    }
}

```

Derived classes can also *override* inherited members by providing an alternate implementation. In order to be able to override a member, the member in the base class must be marked with the `virtual` keyword. By default, base class members are not marked as `virtual` and cannot be overridden. Attempting to override a non-virtual member, as the following example does, generates compiler error CS0506: "<member> cannot override inherited member <member> because it is not marked virtual, abstract, or override.

```

public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}

```

In some cases, a derived class *must* override the base class implementation. Base class members marked with the `abstract` keyword require that derived classes override them. Attempting to compile the following example generates compiler error CS0534, "<class> does not implement inherited abstract member <member>", because class `B` provides no implementation for `A.Method1`.

```

public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}

```

Inheritance applies only to classes and interfaces. Other type categories (structs, delegates, and enums) do not support inheritance. Because of these rules, attempting to compile code like the following example produces compiler error CS0527: "Type 'ValueType' in interface list is not an interface." The error message indicates that, although you can define the interfaces that a struct implements, inheritance is not supported.

```
using System;

public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Implicit inheritance

Besides any types that they may inherit from through single inheritance, all types in the .NET type system implicitly inherit from [Object](#) or a type derived from it. The common functionality of [Object](#) is available to any type.

To see what implicit inheritance means, let's define a new class, `SimpleClass`, that is simply an empty class definition:

```
public class SimpleClass
{ }
```

You can then use reflection (which lets you inspect a type's metadata to get information about that type) to get a list of the members that belong to the `SimpleClass` type. Although you haven't defined any members in your `SimpleClass` class, output from the example indicates that it actually has nine members. One of these members is a parameterless (or default) constructor that is automatically supplied for the `SimpleClass` type by the C# compiler. The remaining eight are members of [Object](#), the type from which all classes and interfaces in the .NET type system ultimately implicitly inherit.

```

using System;
using System.Reflection;

public class Example
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (var member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by
{member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}

// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method): Public, Declared by System.Object
// Equals (Method): Public, Declared by System.Object
// Equals (Method): Public Static, Declared by System.Object
// ReferenceEquals (Method): Public Static, Declared by System.Object
// GetHashCode (Method): Public, Declared by System.Object
// GetType (Method): Public, Declared by System.Object
// Finalize (Method): Internal, Declared by System.Object
// MemberwiseClone (Method): Internal, Declared by System.Object
// .ctor (Constructor): Public, Declared by SimpleClass

```

Implicit inheritance from the `Object` class makes these methods available to the `SimpleClass` class:

- The public `ToString` method, which converts a `SimpleClass` object to its string representation, returns the fully qualified type name. In this case, the `ToString` method returns the string "SimpleClass".
- Three methods that test for equality of two objects: the public instance `Equals(Object)` method, the public static `Equals(Object, Object)` method, and the public static `ReferenceEquals(Object, Object)` method. By default, these methods test for reference equality; that is, to be equal, two object variables must refer to the same object.
- The public `GetHashCode` method, which computes a value that allows an instance of the type to be used in hashed collections.
- The public `GetType` method, which returns a `Type` object that represents the `SimpleClass` type.

- The protected [Finalize](#) method, which is designed to release unmanaged resources before an object's memory is reclaimed by the garbage collector.
- The protected [MemberwiseClone](#) method, which creates a shallow clone of the current object.

Because of implicit inheritance, you can call any inherited member from a `SimpleClass` object just as if it was actually a member defined in the `SimpleClass` class. For instance, the following example calls the `SimpleClass.ToString` method, which `SimpleClass` inherits from [Object](#).

```
using System;

public class SimpleClass
{}

public class Example
{
    public static void Main()
    {
        SimpleClass sc = new SimpleClass();
        Console.WriteLine(sc.ToString());
    }
}

// The example displays the following output:
//      SimpleClass
```

The following table lists the categories of types that you can create in C# and the types from which they implicitly inherit. Each base type makes a different set of members available through inheritance to implicitly derived types.

TYPE CATEGORY	IMPLICITLY INHERITS FROM
class	Object
struct	ValueType , Object
enum	Enum , ValueType , Object
delegate	MulticastDelegate , Delegate , Object

Inheritance and an "is a" relationship

Ordinarily, inheritance is used to express an "is a" relationship between a base class and one or more derived classes, where the derived classes are specialized versions of the base class; the derived class is a type of the base class. For example, the `Publication` class represents a publication of any kind, and the `Book` and `Magazine` classes represent specific types of publications.

NOTE

A class or struct can implement one or more interfaces. While interface implementation is often presented as a workaround for single inheritance or as a way of using inheritance with structs, it is intended to express a different relationship (a "can do" relationship) between an interface and its implementing type than inheritance. An interface defines a subset of functionality (such as the ability to test for equality, to compare or sort objects, or to support culture-sensitive parsing and formatting) that the interface makes available to its implementing types.

Note that "is a" also expresses the relationship between a type and a specific instantiation of that type. In the following example, `Automobile` is a class that has three unique read-only properties: `Make`, the manufacturer of

the automobile; `Model`, the kind of automobile; and `Year`, its year of manufacture. Your `Automobile` class also has a constructor whose arguments are assigned to the property values, and it overrides the `Object.ToString` method to produce a string that uniquely identifies the `Automobile` instance rather than the `Automobile` class.

```
using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException("The make cannot be null.");
        else if (String.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException("The model cannot be null.");
        else if (String.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

In this case, you shouldn't rely on inheritance to represent specific car makes and models. For example, you don't need to define a `Packard` type to represent automobiles manufactured by the Packard Motor Car Company. Instead, you can represent them by creating an `Automobile` object with the appropriate values passed to its class constructor, as the following example does.

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//      1948 Packard Custom Eight
```

An is-a relationship based on inheritance is best applied to a base class and to derived classes that add additional members to the base class or that require additional functionality not present in the base class.

Designing the base class and derived classes

Let's look at the process of designing a base class and its derived classes. In this section, you'll define a base class, `Publication`, which represents a publication of any kind, such as a book, a magazine, a newspaper, a

journal, an article, etc. You'll also define a `Book` class that derives from `Publication`. You could easily extend the example to define other derived classes, such as `Magazine`, `Journal`, `Newspaper`, and `Article`.

The base Publication class

In designing your `Publication` class, you need to make several design decisions:

- What members to include in your base `Publication` class, and whether the `Publication` members provide method implementations or whether `Publication` is an abstract base class that serves as a template for its derived classes.

In this case, the `Publication` class will provide method implementations. The [Designing abstract base classes and their derived classes](#) section contains an example that uses an abstract base class to define the methods that derived classes must override. Derived classes are free to provide any implementation that is suitable for the derived type.

The ability to reuse code (that is, multiple derived classes share the declaration and implementation of base class methods and do not need to override them) is an advantage of non-abstract base classes.

Therefore, you should add members to `Publication` if their code is likely to be shared by some or most specialized `Publication` types. If you fail to provide base class implementations efficiently, you'll end up having to provide largely identical member implementations in derived classes rather a single implementation in the base class. The need to maintain duplicated code in multiple locations is a potential source of bugs.

Both to maximize code reuse and to create a logical and intuitive inheritance hierarchy, you want to be sure that you include in the `Publication` class only the data and functionality that is common to all or to most publications. Derived classes then implement members that are unique to the particular kinds of publication that they represent.

- How far to extend your class hierarchy. Do you want to develop a hierarchy of three or more classes, rather than simply a base class and one or more derived classes? For example, `Publication` could be a base class of `Periodical`, which in turn is a base class of `Magazine`, `Journal` and `Newspaper`.

For your example, you'll use the small hierarchy of a `Publication` class and a single derived class, `Book`. You could easily extend the example to create a number of additional classes that derive from `Publication`, such as `Magazine` and `Article`.

- Whether it makes sense to instantiate the base class. If it does not, you should apply the `abstract` keyword to the class. Otherwise, your `Publication` class can be instantiated by calling its class constructor. If an attempt is made to instantiate a class marked with the `abstract` keyword by a direct call to its class constructor, the C# compiler generates error CS0144, "Cannot create an instance of the abstract class or interface." If an attempt is made to instantiate the class by using reflection, the reflection method throws a [MemberAccessException](#).

By default, a base class can be instantiated by calling its class constructor. You do not have to explicitly define a class constructor. If one is not present in the base class' source code, the C# compiler automatically provides a default (parameterless) constructor.

For your example, you'll mark the `Publication` class as `abstract` so that it cannot be instantiated. An `abstract` class without any `abstract` methods indicates that this class represents an abstract concept that is shared among several concrete classes (like a `Book`, `Journal`).

- Whether derived classes must inherit the base class implementation of particular members, whether they have the option to override the base class implementation, or whether they must provide an implementation. You use the `abstract` keyword to force derived classes to provide an implementation. You use the `virtual` keyword to allow derived classes to override a base class method. By default, methods defined in the base class are *not* overridable.

The `Publication` class does not have any `abstract` methods, but the class itself is `abstract`.

- Whether a derived class represents the final class in the inheritance hierarchy and cannot itself be used as a base class for additional derived classes. By default, any class can serve as a base class. You can apply the `sealed` keyword to indicate that a class cannot serve as a base class for any additional classes. Attempting to derive from a sealed class generates compiler error CS0509, "cannot derive from sealed type <typeName>".

For your example, you'll mark your derived class as `sealed`.

The following example shows the source code for the `Publication` class, as well as a `PublicationType` enumeration that is returned by the `Publication.PublicationType` property. In addition to the members that it inherits from `Object`, the `Publication` class defines the following unique members and member overrides:

```
using System;

public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool published = false;
    private DateTime datePublished;
    private int totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (String.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (String.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException("The number of pages cannot be zero or negative.");
            totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!published)
            return "NYP";
        else
            return datePublished.ToString("d");
    }
}
```

```

public void Publish(DateTime datePublished)
{
    published = true;
    this.datePublished = datePublished;
}

public void Copyright(string copyrightName, int copyrightDate)
{
    if (String.IsNullOrWhiteSpace(copyrightName))
        throw new ArgumentException("The name of the copyright holder is required.");
    CopyrightName = copyrightName;

    int currentYear = DateTime.Now.Year;
    if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
        throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and {currentYear + 1}");
    CopyrightDate = copyrightDate;
}

public override string ToString() => Title;
}

```

- A constructor

Because the `Publication` class is `abstract`, it cannot be instantiated directly from code like the following example:

```

var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
    PublicationType.Book);

```

However, its instance constructor can be called directly from derived class constructors, as the source code for the `Book` class shows.

- Two publication-related properties

`Title` is a read-only `String` property whose value is supplied by calling the `Publication` constructor.

`Pages` is a read-write `Int32` property that indicates how many total pages the publication has. The value is stored in a private field named `totalPages`. It must be a positive number or an `ArgumentOutOfRangeException` is thrown.

- Publisher-related members

Two read-only properties, `Publisher` and `Type`. The values are originally supplied by the call to the `Publication` class constructor.

- Publishing-related members

Two methods, `Publish` and `GetPublicationDate`, set and return the publication date. The `Publish` method sets a private `published` flag to `true` when it is called and assigns the date passed to it as an argument to the private `datePublished` field. The `GetPublicationDate` method returns the string "NYP" if the `published` flag is `false`, and the value of the `datePublished` field if it is `true`.

- Copyright-related members

The `Copyright` method takes the name of the copyright holder and the year of the copyright as arguments and assigns them to the `CopyrightName` and `CopyrightDate` properties.

- An override of the `ToString` method

If a type does not override the `Object.ToString` method, it returns the fully qualified name of the type,

which is of little use in differentiating one instance from another. The `Publication` class overrides `Object.ToString` to return the value of the `Title` property.

The following figure illustrates the relationship between your base `Publication` class and its implicitly inherited `Object` class.

Object	Publication
Equals(Object)	Equals(Object)
Equals(Object, Object)	Equals(Object, Object)
Finalize()	Finalize()
GetHashCode()	GetHashCode()
GetType()	GetType()
MemberwiseClone()	MemberwiseClone()
ReferenceEquals()	ReferenceEquals()
ToString()	ToString()
#ctor()	#ctor(String, String, PublicationType)
	PublicationType
	Publisher
	Title
	CopyrightDate
	CopyrightName
	Pages
	Copyright()
	GetPublicationDate()
	Publish()

Key

Unique member	
Inherited member	
Overridden member	

The `Book` class

The `Book` class represents a book as a specialized type of publication. The following example shows the source code for the `Book` class.

```

using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
        this(title, String.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher) : base(title, publisher,
PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string without "-" characters.
        // We could also determine whether the ISBN is valid by comparing its checksum digit
        // with a computed checksum.
        //
        if (! String.IsNullOrEmpty(isbn)) {
            // Determine if ISBN length is correct.
            if (! (isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-character numeric string.");
            ulong nISBN = 0;
            if (! UInt64.TryParse(isbn, out nISBN))
                throw new ArgumentException("The ISBN can consist of numeric characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public Decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public Decimal SetPrice(Decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException("The price cannot be negative.");
        Decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object obj)
    {
        Book book = obj as Book;
        if (book == null)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{{({String.IsNullOrEmpty(Author) ? "" : Author + ", "}){Title}}}";
}

```

In addition to the members that it inherits from `Publication`, the `Book` class defines the following unique members and member overrides:

- Two constructors

The two `Book` constructors share three common parameters. Two, `title` and `publisher`, correspond to parameters of the `Publication` constructor. The third is `author`, which is stored to a public immutable `Author` property. One constructor includes an `isbn` parameter, which is stored in the `ISBN` auto-property.

The first constructor uses the `this` keyword to call the other constructor. Constructor chaining is a common pattern in defining constructors. Constructors with fewer parameters provide default values when calling the constructor with the greatest number of parameters.

The second constructor uses the `base` keyword to pass the title and publisher name to the base class constructor. If you don't make an explicit call to a base class constructor in your source code, the C# compiler automatically supplies a call to the base class' default or parameterless constructor.

- A read-only `ISBN` property, which returns the `Book` object's International Standard Book Number, a unique 10- or 13-digit number. The ISBN is supplied as an argument to one of the `Book` constructors. The ISBN is stored in a private backing field, which is auto-generated by the compiler.
- A read-only `Author` property. The author name is supplied as an argument to both `Book` constructors and is stored in the property.
- Two read-only price-related properties, `Price` and `Currency`. Their values are provided as arguments in a `SetPrice` method call. The `Currency` property is the three-digit ISO currency symbol (for example, USD for the U.S. dollar). ISO currency symbols can be retrieved from the `ISOCurrencySymbol` property. Both of these properties are externally read-only, but both can be set by code in the `Book` class.
- A `SetPrice` method, which sets the values of the `Price` and `Currency` properties. Those values are returned by those same properties.
- Overrides to the `ToString` method (inherited from `Publication`) and the `Object.Equals(Object)` and `GetHashCode` methods (inherited from `Object`).

Unless it is overridden, the `Object.Equals(Object)` method tests for reference equality. That is, two object variables are considered to be equal if they refer to the same object. In the `Book` class, on the other hand, two `Book` objects should be equal if they have the same ISBN.

When you override the `Object.Equals(Object)` method, you must also override the `GetHashCode` method, which returns a value that the runtime uses to store items in hashed collections for efficient retrieval. The hash code should return a value that's consistent with the test for equality. Since you've overridden `Object.Equals(Object)` to return `true` if the ISBN properties of two `Book` objects are equal, you return the hash code computed by calling the `GetHashCode` method of the string returned by the `ISBN` property.

The following figure illustrates the relationship between the `Book` class and `Publication`, its base class.

Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

Book

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

Key

Unique member	
Inherited member	
Overridden member	

You can now instantiate a `Book` object, invoke both its unique and inherited members, and pass it as an argument to a method that expects a parameter of type `Publication` or of type `Book`, as the following example shows.

```

using System;
using static System.Console;

public class Example
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare, William",
                            "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare, William");
        WriteLine($"{book.Title} and {book2.Title} are the same publication: " +
                 $""\{((Publication) book).Equals(book2)}\"");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        WriteLine($"{pub.Title}, " +
                  $""\{(pubDate == "NYP" ? "Not Yet Published" : "published on " + pubDate):d} by
{pub.Publisher}"); 
    }
}

// The example displays the following output:
//      The Tempest, Not Yet Published by Public Domain Press
//      The Tempest, published on 8/18/2016 by Public Domain Press
//      The Tempest and The Tempest are the same publication: False

```

Designing abstract base classes and their derived classes

In the previous example, you defined a base class that provided an implementation for a number of methods to allow derived classes to share code. In many cases, however, the base class is not expected to provide an implementation. Instead, the base class is an *abstract class* that declares *abstract methods*; it serves as a template that defines the members that each derived class must implement. Typically in an abstract base class, the implementation of each derived type is unique to that type. You marked the class with the `abstract` keyword because it made no sense to instantiate a `Publication` object, although the class did provide implementations of functionality common to publications.

For example, each closed two-dimensional geometric shape includes two properties: area, the inner extent of the shape; and perimeter, or the distance along the edges of the shape. The way in which these properties are calculated, however, depends completely on the specific shape. The formula for calculating the perimeter (or circumference) of a circle, for example, is different from that of a triangle. The `Shape` class is an `abstract` class with `abstract` methods. That indicates derived classes share the same functionality, but those derived classes implement that functionality differently.

The following example defines an abstract base class named `Shape` that defines two properties: `Area` and `Perimeter`. In addition to marking the class with the `abstract` keyword, each instance member is also marked with the `abstract` keyword. In this case, `Shape` also overrides the `Object.ToString` method to return the name of the type, rather than its fully qualified name. And it defines two static members, `GetArea` and `GetPerimeter`, that allow callers to easily retrieve the area and perimeter of an instance of any derived class. When you pass an instance of a derived class to either of these methods, the runtime calls the method override of the derived class.

```
using System;

public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

You can then derive some classes from `Shape` that represent specific shapes. The following example defines three classes, `Triangle`, `Rectangle`, and `Circle`. Each uses a formula unique for that particular shape to compute the area and perimeter. Some of the derived classes also define properties, such as `Rectangle.Diagonal` and `Circle.Diameter`, that are unique to the shape that they represent.

```

using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) + Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}

```

The following example uses objects derived from `Shape`. It instantiates an array of objects derived from `Shape` and calls the static methods of the `Shape` class, which wraps return `Shape` property values. The runtime retrieves values from the overridden properties of the derived types. The example also casts each `Shape` object in the array to its derived type and, if the cast succeeds, retrieves properties of that particular subclass of `Shape`.

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (var shape in shapes) {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                $"perimeter, {Shape.GetPerimeter(shape)}");
            var rect = shape as Rectangle;
            if (rect != null) {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
                continue;
            }
            var sq = shape as Square;
            if (sq != null) {
                Console.WriteLine($"    Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}

// The example displays the following output:
//     Rectangle: area, 120; perimeter, 44
//             Is Square: False, Diagonal: 15.62
//     Square: area, 25; perimeter, 20
//             Diagonal: 7.07
//     Circle: area, 28.27; perimeter, 18.85
```

See also

- [Inheritance \(C# Programming Guide\)](#)

Work with Language-Integrated Query (LINQ)

3/23/2021 • 15 minutes to read • [Edit Online](#)

Introduction

This tutorial teaches you features in .NET Core and the C# language. You'll learn how to:

- Generate sequences with LINQ.
- Write methods that can be easily used in LINQ queries.
- Distinguish between eager and lazy evaluation.

You'll learn these techniques by building an application that demonstrates one of the basic skills of any magician: the [faro shuffle](#). Briefly, a faro shuffle is a technique where you split a card deck exactly in half, then the shuffle interleaves each one card from each half to rebuild the original deck.

Magicians use this technique because every card is in a known location after each shuffle, and the order is a repeating pattern.

For your purposes, it is a light hearted look at manipulating sequences of data. The application you'll build constructs a card deck and then performs a sequence of shuffles, writing the sequence out each time. You'll also compare the updated order to the original order.

This tutorial has multiple steps. After each step, you can run the application and see the progress. You can also see the [completed sample](#) in the `dotnet/samples` GitHub repository. For download instructions, see [Samples and Tutorials](#).

Prerequisites

You'll need to set up your machine to run .NET core. You can find the installation instructions on the [.NET Core Download](#) page. You can run this application on Windows, Ubuntu Linux, or OS X, or in a Docker container. You'll need to install your favorite code editor. The descriptions below use [Visual Studio Code](#) which is an open source, cross-platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

If you've never used C# before, [this tutorial](#) explains the structure of a C# program. You can read that and then return here to learn more about LINQ.

Create the Data Set

Before you begin, make sure that the following lines are at the top of the `Program.cs` file generated by `dotnet new console`:

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

If these three lines (`using` statements) aren't at the top of the file, our program will not compile.

Now that you have all of the references that you'll need, consider what constitutes a deck of cards. Commonly, a deck of playing cards has four suits, and each suit has thirteen values. Normally, you might consider creating a `Card` class right off the bat and populating a collection of `Card` objects by hand. With LINQ, you can be more concise than the usual way of dealing with creating a deck of cards. Instead of creating a `Card` class, you can create two sequences to represent suits and ranks, respectively. You'll create a really simple pair of [iterator methods](#) that will generate the ranks and suits as `IEnumerable<T>`s of strings:

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Place these underneath the `Main` method in your `Program.cs` file. These two methods both utilize the `yield return` syntax to produce a sequence as they run. The compiler builds an object that implements `IEnumerable<T>` and generates the sequence of strings as they are requested.

Now, use these iterator methods to create the deck of cards. You'll place the LINQ query in our `Main` method. Here's a look at it:

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

The multiple `from` clauses produce a [SelectMany](#), which creates a single sequence from combining each element in the first sequence with each element in the second sequence. The order is important for our purposes. The first element in the first source sequence (Suits) is combined with every element in the second

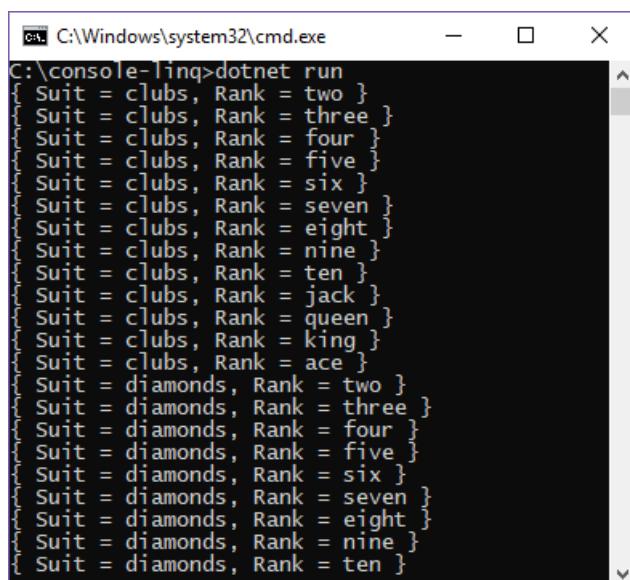
sequence (Ranks). This produces all thirteen cards of first suit. That process is repeated with each element in the first sequence (Suits). The end result is a deck of cards ordered by suits, followed by values.

It's important to keep in mind that whether you choose to write your LINQ in the query syntax used above or use method syntax instead, it's always possible to go from one form of syntax to the other. The above query written in query syntax can be written in method syntax as:

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new { Suit = suit, Rank = rank }));
```

The compiler translates LINQ statements written with query syntax into the equivalent method call syntax. Therefore, regardless of your syntax choice, the two versions of the query produce the same result. Choose which syntax works best for your situation: for instance, if you're working in a team where some of the members have difficulty with method syntax, try to prefer using query syntax.

Go ahead and run the sample you've built at this point. It will display all 52 cards in the deck. You may find it very helpful to run this sample under a debugger to observe how the `Suits()` and `Ranks()` methods execute. You can clearly see that each string in each sequence is generated only as it is needed.



```
C:\console-linq>dotnet run
{ Suit = clubs, Rank = two }
{ Suit = clubs, Rank = three }
{ Suit = clubs, Rank = four }
{ Suit = clubs, Rank = five }
{ Suit = clubs, Rank = six }
{ Suit = clubs, Rank = seven }
{ Suit = clubs, Rank = eight }
{ Suit = clubs, Rank = nine }
{ Suit = clubs, Rank = ten }
{ Suit = clubs, Rank = jack }
{ Suit = clubs, Rank = queen }
{ Suit = clubs, Rank = king }
{ Suit = clubs, Rank = ace }
{ Suit = diamonds, Rank = two }
{ Suit = diamonds, Rank = three }
{ Suit = diamonds, Rank = four }
{ Suit = diamonds, Rank = five }
{ Suit = diamonds, Rank = six }
{ Suit = diamonds, Rank = seven }
{ Suit = diamonds, Rank = eight }
{ Suit = diamonds, Rank = nine }
{ Suit = diamonds, Rank = ten }
```

Manipulate the Order

Next, focus on how you're going to shuffle the cards in the deck. The first step in any good shuffle is to split the deck in two. The `Take` and `Skip` methods that are part of the LINQ APIs provide that feature for you. Place them underneath the `foreach` loop:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

However, there's no shuffle method to take advantage of in the standard library, so you'll have to write your own. The shuffle method you'll be creating illustrates several techniques that you'll use with LINQ-based programs, so each part of this process will be explained in steps.

In order to add some functionality to how you interact with the `IEnumerable<T>` you'll get back from LINQ queries, you'll need to write some special kinds of methods called [extension methods](#). Briefly, an extension method is a special purpose *static method* that adds new functionality to an already-existing type without having to modify the original type you want to add functionality to.

Give your extension methods a new home by adding a new *static* class file to your program called `Extensions.cs`, and then start building out the first extension method:

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this IEnumerable<T> first, IEnumerable<T> second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Look at the method signature for a moment, specifically the parameters:

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T> first, IEnumerable<T> second)
```

You can see the addition of the `this` modifier on the first argument to the method. That means you call the method as though it were a member method of the type of the first argument. This method declaration also follows a standard idiom where the input and output types are `IEnumerable<T>`. That practice enables LINQ methods to be chained together to perform more complex queries.

Naturally, since you split the deck into halves, you'll need to join those halves together. In code, this means you'll be enumerating both of the sequences you acquired through `Take` and `Skip` at once, `interleaving` the elements, and creating one sequence: your now-shuffled deck of cards. Writing a LINQ method that works with two sequences requires that you understand how `IEnumerable<T>` works.

The `IEnumerable<T>` interface has one method: `GetEnumerator`. The object returned by `GetEnumerator` has a method to move to the next element, and a property that retrieves the current element in the sequence. You will use those two members to enumerate the collection and return the elements. This Interleave method will be an iterator method, so instead of building a collection and returning the collection, you'll use the `yield return` syntax shown above.

Here's the implementation of that method:

```

public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}

```

Now that you've written this method, go back to the `Main` method and shuffle the deck once:

```

// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                        from r in Ranks()
                        select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}

```

Comparisons

How many shuffles it takes to set the deck back to its original order? To find out, you'll need to write a method that determines if two sequences are equal. After you have that method, you'll need to place the code that shuffles the deck in a loop, and check to see when the deck is back in order.

Writing a method to determine if the two sequences are equal should be straightforward. It's a similar structure to the method you wrote to shuffle the deck. Only this time, instead of `yield return`ing each element, you'll compare the matching elements of each sequence. When the entire sequence has been enumerated, if every element matches, the sequences are the same:

```

public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}

```

This shows a second LINQ idiom: terminal methods. They take a sequence as input (or in this case, two sequences), and return a single scalar value. When using terminal methods, they are always the final method in a chain of methods for a LINQ query, hence the name "terminal".

You can see this in action when you use it to determine when the deck is back in its original order. Put the shuffle code inside a loop, and stop when the sequence is back in its original order by applying the `SequenceEquals()` method. You can see it would always be the final method in any query, because it returns a single value instead of a sequence:

```

// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}

```

Run the code you've got so far and take note of how the deck rearranges on each shuffle. After 8 shuffles (iterations of the do-while loop), the deck returns to the original configuration it was in when you first created it from the starting LINQ query.

Optimizations

The sample you've built so far executes an *out shuffle*, where the top and bottom cards stay the same on each run. Let's make one change: we'll use an *in shuffle* instead, where all 52 cards change position. For an in shuffle, you interleave the deck so that the first card in the bottom half becomes the first card in the deck. That means

the last card in the top half becomes the bottom card. This is a simple change to a singular line of code. Update the current shuffle query by switching the positions of [Take](#) and [Skip](#). This will change the order of the top and bottom halves of the deck:

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Run the program again, and you'll see that it takes 52 iterations for the deck to reorder itself. You'll also start to notice some serious performance degradations as the program continues to run.

There are a number of reasons for this. You can tackle one of the major causes of this performance drop: inefficient use of [lazy evaluation](#).

Briefly, lazy evaluation states that the evaluation of a statement is not performed until its value is needed. LINQ queries are statements that are evaluated lazily. The sequences are generated only as the elements are requested. Usually, that's a major benefit of LINQ. However, in a use such as this program, this causes exponential growth in execution time.

Remember that we generated the original deck using a LINQ query. Each shuffle is generated by performing three LINQ queries on the previous deck. All these are performed lazily. That also means they are performed again each time the sequence is requested. By the time you get to the 52nd iteration, you're regenerating the original deck many, many times. Let's write a log to demonstrate this behavior. Then, you'll fix it.

In your `Extensions.cs` file, type in or copy the method below. This extension method creates a new file called `debug.log` within your project directory and records what query is currently being executed to the log file. This extension method can be appended to any query to mark that the query executed.

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

You will see a red squiggle under `File`, meaning it doesn't exist. It won't compile, since the compiler doesn't know what `File` is. To solve this problem, make sure to add the following line of code under the very first line in `Extensions.cs`:

```
using System.IO;
```

This should solve the issue and the red error disappears.

Next, instrument the definition of each query with a log message:

```

// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r }).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26))
            .LogQuery("Bottom Half")
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}

```

Notice that you don't log every time you access a query. You log only when you create the original query. The program still takes a long time to run, but now you can see why. If you run out of patience running the in shuffle with logging turned on, switch back to the out shuffle. You'll still see the lazy evaluation effects. In one run, it executes 2592 queries, including all the value and suit generation.

You can improve the performance of the code here to reduce the number of executions you make. A simple fix you can make is to *cache* the results of the original LINQ query that constructs the deck of cards. Currently, you're executing the queries again and again every time the do-while loop goes through an iteration, reconstructing the deck of cards and reshuffling it every time. To cache the deck of cards, you can leverage the LINQ methods [ToArray](#) and [ToList](#); when you append them to the queries, they'll perform the same actions you've told them to, but now they'll store the results in an array or a list, depending on which method you choose to call. Append the LINQ method [ToArray](#) to both queries and run the program again:

```

public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}

```

Now the out shuffle is down to 30 queries. Run again with the in shuffle and you'll see similar improvements: it now executes 162 queries.

Please note that this example is **designed** to highlight the use cases where lazy evaluation can cause performance difficulties. While it's important to see where lazy evaluation can impact code performance, it's equally important to understand that not all queries should run eagerly. The performance hit you incur without using [ToArray](#) is because each new arrangement of the deck of cards is built from the previous arrangement. Using lazy evaluation means each new deck configuration is built from the original deck, even executing the code that built the `startingDeck`. That causes a large amount of extra work.

In practice, some algorithms run well using eager evaluation, and others run well using lazy evaluation. For daily usage, lazy evaluation is usually a better choice when the data source is a separate process, like a database engine. For databases, lazy evaluation allows more complex queries to execute only one round trip to the database process and back to the rest of your code. LINQ is flexible whether you choose to utilize lazy or eager evaluation, so measure your processes and pick whichever kind of evaluation gives you the best performance.

Conclusion

In this project, you covered:

- using LINQ queries to aggregate data into a meaningful sequence
- writing Extension methods to add our own custom functionality to LINQ queries
- locating areas in our code where our LINQ queries might run into performance issues like degraded speed
- lazy and eager evaluation in regards to LINQ queries and the implications they might have on query performance

Aside from LINQ, you learned a bit about a technique magicians use for card tricks. Magicians use the Faro shuffle because they can control where every card moves in the deck. Now that you know, don't spoil it for everyone else!

For more information on LINQ, see:

- [Language Integrated Query \(LINQ\)](#)
- [Introduction to LINQ](#)
- [Basic LINQ Query Operations \(C#\)](#)
- [Data Transformations With LINQ \(C#\)](#)
- [Query Syntax and Method Syntax in LINQ \(C#\)](#)
- [C# Features That Support LINQ](#)

Use Attributes in C#

11/2/2020 • 7 minutes to read • [Edit Online](#)

Attributes provide a way of associating information with code in a declarative way. They can also provide a reusable element that can be applied to a variety of targets.

Consider the `[Obsolete]` attribute. It can be applied to classes, structs, methods, constructors, and more. It *declares* that the element is obsolete. It's then up to the C# compiler to look for this attribute, and do some action in response.

In this tutorial, you'll be introduced to how to add attributes to your code, how to create and use your own attributes, and how to use some attributes that are built into .NET Core.

Prerequisites

You'll need to set up your machine to run .NET core. You can find the installation instructions on the [.NET Core Downloads](#) page. You can run this application on Windows, Ubuntu Linux, macOS or in a Docker container. You'll need to install your favorite code editor. The descriptions below use [Visual Studio Code](#) which is an open source, cross platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

Now that you've installed all the tools, create a new .NET Core application. To use the command line generator, execute the following command in your favorite shell:

```
dotnet new console
```

This command will create bare-bones .NET core project files. You will need to execute `dotnet restore` to restore the dependencies needed to compile this project.

You don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore documentation](#).

To execute the program, use `dotnet run`. You should see "Hello, World" output to the console.

How to add attributes to code

In C#, attributes are classes that inherit from the `Attribute` base class. Any class that inherits from `Attribute` can be used as a sort of "tag" on other pieces of code. For instance, there is an attribute called `ObsoleteAttribute`. This is used to signal that code is obsolete and shouldn't be used anymore. You can place this attribute on a class, for instance, by using square brackets.

```
[Obsolete]
public class MyClass
{
}
```

Note that while the class is called `ObsoleteAttribute`, it's only necessary to use `[Obsolete]` in the code. This is a convention that C# follows. You can use the full name `[ObsoleteAttribute]` if you choose.

When marking a class obsolete, it's a good idea to provide some information as to *why* it's obsolete, and/or *what* to use instead. Do this by passing a string parameter to the `Obsolete` attribute.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

The string is being passed as an argument to an `ObsoleteAttribute` constructor, just as if you were writing `var attr = new ObsoleteAttribute("some string")`.

Parameters to an attribute constructor are limited to simple types/literals:

`bool, int, double, string, Type, enums, etc` and arrays of those types. You can not use an expression or a variable. You are free to use positional or named parameters.

How to create your own attribute

Creating an attribute is as simple as inheriting from the `Attribute` base class.

```
public class MySpecialAttribute : Attribute
{
}
```

With the above, I can now use `[MySpecial]` (or `[MySpecialAttribute]`) as an attribute elsewhere in the code base.

```
[MySpecial]
public class SomeOtherClass
{
}
```

Attributes in the .NET base class library like `ObsoleteAttribute` trigger certain behaviors within the compiler. However, any attribute you create acts only as metadata, and doesn't result in any code within the attribute class being executed. It's up to you to act on that metadata elsewhere in your code (more on that later in the tutorial).

There is a 'gotcha' here to watch out for. As mentioned above, only certain types are allowed to be passed as arguments when using attributes. However, when creating an attribute type, the C# compiler won't stop you from creating those parameters. In the below example, I've created an attribute with a constructor that compiles just fine.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {
    }
}
```

However, you will be unable to use this constructor with attribute syntax.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

The above will cause a compiler error like

```
Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type
```

How to restrict attribute usage

Attributes can be used on a number of "targets". The above examples show them on classes, but they can also be used on:

- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

When you create an attribute class, by default, C# will allow you to use that attribute on any of the possible attribute targets. If you want to restrict your attribute to certain targets, you can do so by using the

```
AttributeUsageAttribute
```

on your attribute class. That's right, an attribute on an attribute!

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{}
```

If you attempt to put the above attribute on something that's not a class or a struct, you will get a compiler error like

```
Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on
'class, struct' declarations
```

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    {
    }
}
```

How to use attributes attached to a code element

Attributes act as metadata. Without some outward force, they won't actually do anything.

To find and act on attributes, [Reflection](#) is generally needed. I won't cover Reflection in-depth in this tutorial, but the basic idea is that Reflection allows you to write code in C# that examines other code.

For instance, you can use Reflection to get information about a class (add `using System.Reflection;` at the head of your code):

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

That will print out something like:

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

Once you have a `TypeInfo` object (or a `MethodInfo`, `FieldInfo`, etc), you can use the `GetCustomAttributes` method. This will return a collection of `Attribute` objects. You can also use `GetCustomAttribute` and specify an Attribute type.

Here's an example of using `GetCustomAttributes` on a `MethodInfo` instance for `MyClass` (which we saw earlier has an `[Obsolete]` attribute on it).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

That will print to console: `Attribute on MyClass: ObsoleteAttribute`. Try adding other attributes to `MyClass`.

It's important to note that these `Attribute` objects are instantiated lazily. That is, they won't be instantiated until you use `GetCustomAttribute` or `GetCustomAttributes`. They are also instantiated each time. Calling `GetCustomAttributes` twice in a row will return two different instances of `ObsoleteAttribute`.

Common attributes in the base class library (BCL)

Attributes are used by many tools and frameworks. NUnit uses attributes like `[Test]` and `[TestFixture]` that are used by the NUnit test runner. ASP.NET MVC uses attributes like `[Authorize]` and provides an action filter framework to perform cross-cutting concerns on MVC actions. [PostSharp](#) uses the attribute syntax to allow aspect-oriented programming in C#.

Here are a few notable attributes built into the .NET Core base class libraries:

- `[Obsolete]`. This one was used in the above examples, and it lives in the `System` namespace. It is useful to provide declarative documentation about a changing code base. A message can be provided in the form of a string, and another boolean parameter can be used to escalate from a compiler warning to a compiler error.
- `[Conditional]`. This attribute is in the `System.Diagnostics` namespace. This attribute can be applied to methods (or attribute classes). You must pass a string to the constructor. If that string doesn't match a `#define` directive, then any calls to that method (but not the method itself) will be removed by the C# compiler. Typically this is used for debugging (diagnostics) purposes.
- `[CallerMemberName]`. This attribute can be used on parameters, and lives in the `System.Runtime.CompilerServices` namespace. This is an attribute that is used to inject the name of the method that is calling another method. This is typically used as a way to eliminate 'magic strings' when

implementing `INotifyPropertyChanged` in various UI frameworks. As an example:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged(); // notice that "Name" is not needed here explicitly
            }
        }
    }
}
```

In the above code, you don't have to have a literal `"Name"` string. This can help prevent typo-related bugs and also makes for smoother refactoring/rename.

Summary

Attributes bring declarative power to C#, but they are a meta-data form of code and don't act by themselves.

Tutorial: Use pattern matching to build type-driven and data-driven algorithms

3/23/2021 • 15 minutes to read • [Edit Online](#)

C# 7 introduced basic pattern matching features. Those features are extended in C# 8 and C# 9 with new expressions and patterns. You can write functionality that behaves as though you extended types that may be in other libraries. Another use for patterns is to create functionality your application requires that isn't a fundamental feature of the type being extended.

In this tutorial, you'll learn how to:

- Recognize situations where pattern matching should be used.
- Use pattern matching expressions to implement behavior based on types and property values.
- Combine pattern matching with other techniques to create complete algorithms.

Prerequisites

You'll need to set up your machine to run .NET 5, which includes the C# 9 compiler. The C# 9 compiler is available starting with [Visual Studio 2019 version 16.9 preview 1](#) or [.NET 5.0 SDK](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Scenarios for pattern matching

Modern development often includes integrating data from multiple sources and presenting information and insights from that data in a single cohesive application. You and your team won't have control or access for all the types that represent the incoming data.

The classic object-oriented design would call for creating types in your application that represent each data type from those multiple data sources. Then, your application would work with those new types, build inheritance hierarchies, create virtual methods, and implement abstractions. Those techniques work, and sometimes they are the best tools. Other times you can write less code. You can write more clear code using techniques that separate the data from the operations that manipulate that data.

In this tutorial, you'll create and explore an application that takes incoming data from several external sources for a single scenario. You'll see how **pattern matching** provides an efficient way to consume and process that data in ways that weren't part of the original system.

Consider a major metropolitan area that is using tolls and peak time pricing to manage traffic. You write an application that calculates tolls for a vehicle based on its type. Later enhancements incorporate pricing based on the number of occupants in the vehicle. Further enhancements add pricing based on the time and the day of the week.

From that brief description, you may have quickly sketched out an object hierarchy to model this system. However, your data is coming from multiple sources like other vehicle registration management systems. These systems provide different classes to model that data and you don't have a single object model you can use. In this tutorial, you'll use these simplified classes to model for the vehicle data from these external systems, as shown in the following code:

```

namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}

```

You can download the starter code from the [dotnet/samples](#) GitHub repository. You can see that the vehicle classes are from different systems, and are in different namespaces. No common base class, other than `System.Object` can be leveraged.

Pattern matching designs

The scenario used in this tutorial highlights the kinds of problems that pattern matching is well suited to solve:

- The objects you need to work with aren't in an object hierarchy that matches your goals. You may be working with classes that are part of unrelated systems.
- The functionality you're adding isn't part of the core abstraction for these classes. The toll paid by a vehicle *changes* for different types of vehicles, but the toll isn't a core function of the vehicle.

When the *shape* of the data and the *operations* on that data are not described together, the pattern matching features in C# make it easier to work with.

Implement the basic toll calculations

The most basic toll calculation relies only on the vehicle type:

- A `Car` is \$2.00.
- A `Taxi` is \$3.50.
- A `Bus` is \$5.00.
- A `DeliveryTruck` is \$10.00

Create a new `TollCalculator` class, and implement pattern matching on the vehicle type to get the toll amount.

The following code shows the initial implementation of the `TollCalculator`.

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    public class TollCalculator
    {
        public decimal CalculateToll(object vehicle) =>
            vehicle switch
            {
                Car c           => 2.00m,
                Taxi t          => 3.50m,
                Bus b           => 5.00m,
                DeliveryTruck t => 10.00m,
                { }              => throw new ArgumentException(message: "Not a known vehicle type", paramName:
nameof(vehicle)),
                null            => throw new ArgumentNullException(nameof(vehicle))
            };
    }
}

```

The preceding code uses a **switch expression** (not the same as a `switch` statement) that tests the **type pattern**. A **switch expression** begins with the variable, `vehicle` in the preceding code, followed by the `switch` keyword. Next comes all the **switch arms** inside curly braces. The `switch` expression makes other refinements to the syntax that surrounds the `switch` statement. The `case` keyword is omitted, and the result of each arm is an expression. The last two arms show a new language feature. The `{ }` case matches any non-null object that didn't match an earlier arm. This arm catches any incorrect types passed to this method. The `{ }` case must follow the cases for each vehicle type. If the order were reversed, the `{ }` case would take precedence. Finally, the `null` pattern detects when a `null` is passed to this method. The `null` pattern can be last because the other type patterns match only a non-null object of the correct type.

You can test this code using the following code in `Program.cs`:

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            var tollCalc = new TollCalculator();

            var car = new Car();
            var taxi = new Taxi();
            var bus = new Bus();
            var truck = new DeliveryTruck();

            Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
            Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
            Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
            Console.WriteLine($"The toll for a truck is {tollCalc.CalculateToll(truck)}");

            try
            {
                tollCalc.CalculateToll("this will fail");
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Caught an argument exception when using the wrong type");
            }
            try
            {
                tollCalc.CalculateToll(null!);
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("Caught an argument exception when using null");
            }
        }
    }
}

```

That code is included in the starter project, but is commented out. Remove the comments, and you can test what you've written.

You're starting to see how patterns can help you create algorithms where the code and the data are separate.

The `switch` expression tests the type and produces different values based on the results. That's only the beginning.

Add occupancy pricing

The toll authority wants to encourage vehicles to travel at maximum capacity. They've decided to charge more when vehicles have fewer passengers, and encourage full vehicles by offering lower pricing:

- Cars and taxis with no passengers pay an extra \$0.50.
- Cars and taxis with two passengers get a \$0.50 discount.
- Cars and taxis with three or more passengers get a \$1.00 discount.
- Buses that are less than 50% full pay an extra \$2.00.
- Buses that are more than 90% full get a \$1.00 discount.

These rules can be implemented using the **property pattern** in the same switch expression. A property pattern

is a `when` clause that compares a property value to a constant value. The property pattern examines properties of the object once the type has been determined. The single case for a `car` expands to four different cases:

```
vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car c                   => 2.00m - 1.0m,

    // ...
};
```

The first three cases test the type as a `Car`, then check the value of the `Passengers` property. If both match, that expression is evaluated and returned.

You would also expand the cases for taxis in a similar manner:

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi t           => 3.50m - 1.00m,

    // ...
};
```

In the preceding example, the `when` clause was omitted on the final case.

Next, implement the occupancy rules by expanding the cases for buses, as shown in the following example:

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    // ...
};
```

The toll authority isn't concerned with the number of passengers in the delivery trucks. Instead, they adjust the toll amount based on the weight class of the trucks as follows:

- Trucks over 5000 lbs are charged an extra \$5.00.
- Light trucks under 3000 lbs are given a \$2.00 discount.

That rule is implemented with the following code:

```

vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,
};

}

```

The preceding code shows the `when` clause of a switch arm. You use the `when` clause to test conditions other than equality on a property. When you've finished, you'll have a method that looks much like the following code:

```

vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car c                   => 2.00m - 1.0m,

    Taxi {Fares: 0}   => 3.50m + 1.00m,
    Taxi {Fares: 1}   => 3.50m,
    Taxi {Fares: 2}   => 3.50m - 0.50m,
    Taxi t            => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,

    { }      => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};

}

```

Many of these switch arms are examples of **recursive patterns**. For example, `Car { Passengers: 1}` shows a constant pattern inside a property pattern.

You can make this code less repetitive by using nested switches. The `Car` and `Taxi` both have four different arms in the preceding examples. In both cases, you can create a type pattern that feeds into a property pattern. This technique is shown in the following code:

```

public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },
        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },
        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
        Bus b => 5.00m,
        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,
        { } => throw new ArgumentException(message: "Not a known vehicle type", paramName:
            nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };

```

In the preceding sample, using a recursive expression means you don't repeat the `Car` and `Taxi` arms containing child arms that test the property value. This technique isn't used for the `Bus` and `DeliveryTruck` arms because those arms are testing ranges for the property, not discrete values.

Add peak pricing

For the final feature, the toll authority wants to add time sensitive peak pricing. During the morning and evening rush hours, the tolls are doubled. That rule only affects traffic in one direction: inbound to the city in the morning, and outbound in the evening rush hour. During other times during the workday, tolls increase by 50%. Late night and early morning, tolls are reduced by 25%. During the weekend, it's the normal rate, regardless of the time. You could use a series of `if` and `else` statements to express this using the following code:

```

public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}

```

The preceding code does work correctly, but isn't readable. You have to chain through all the input cases and the nested `if` statements to reason about the code. Instead, you'll use pattern matching for this feature, but you'll integrate it with other techniques. You could build a single pattern match expression that would account for all the combinations of direction, day of the week, and time. The result would be a complicated expression. It would be hard to read and difficult to understand. That makes it hard to ensure correctness. Instead, combine those methods to build a tuple of values that concisely describes all those states. Then use pattern matching to calculate a multiplier for the toll. The tuple contains three discrete conditions:

- The day is either a weekday or a weekend.
- The band of time when the toll is collected.
- The direction is into the city or out of the city

The following table shows the combinations of input values and the peak pricing multiplier:

DAY	TIME	DIRECTION	PREMIUM
Weekday	morning rush	inbound	x 2.00
Weekday	morning rush	outbound	x 1.00
Weekday	daytime	inbound	x 1.50
Weekday	daytime	outbound	x 1.50
Weekday	evening rush	inbound	x 1.00
Weekday	evening rush	outbound	x 2.00
Weekday	overnight	inbound	x 0.75
Weekday	overnight	outbound	x 0.75
Weekend	morning rush	inbound	x 1.00
Weekend	morning rush	outbound	x 1.00
Weekend	daytime	inbound	x 1.00
Weekend	daytime	outbound	x 1.00
Weekend	evening rush	inbound	x 1.00
Weekend	evening rush	outbound	x 1.00
Weekend	overnight	inbound	x 1.00
Weekend	overnight	outbound	x 1.00

There are 16 different combinations of the three variables. By combining some of the conditions, you'll simplify the final switch expression.

The system that collects the tolls uses a [DateTime](#) structure for the time when the toll was collected. Build member methods that create the variables from the preceding table. The following function uses a pattern matching switch expression to express whether a [DateTime](#) represents a weekend or a weekday:

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

That method is correct, but it's repetitious. You can simplify it, as shown in the following code:

```

private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };

```

Next, add a similar function to categorize the time into the blocks:

```

private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.OVERNIGHT,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };

```

You add a private `enum` to convert each range of time to a discrete value. Then, the `GetTimeBand` method uses *relational patterns*, and *conjunctive or patterns*, both added in C# 9.0. The relational pattern lets you test a numeric value using `<`, `>`, `<=`, or `>=`. The `or` pattern tests if an expression matches one or more patterns. You can also use an `and` pattern to ensure that an expression matches two distinct patterns, and a `not` pattern to test that an expression doesn't match a pattern.

After you create those methods, you can use another `switch` expression with the *tuple pattern* to calculate the pricing premium. You could build a `switch` expression with all 16 arms:

```

public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.OVERNIGHT, true) => 0.75m,
        (true, TimeBand.OVERNIGHT, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.OVERNIGHT, true) => 1.00m,
        (false, TimeBand.OVERNIGHT, false) => 1.00m,
    };

```

The above code works, but it can be simplified. All eight combinations for the weekend have the same toll. You can replace all eight with the following line:

```
(false, _, _) => 1.0m,
```

Both inbound and outbound traffic have the same multiplier during the weekday daytime and overnight hours. Those four switch arms can be replaced with the following two lines:

```
(true, TimeBand.Overnight, _) => 0.75m,  
(true, TimeBand.Daytime, _)    => 1.5m,
```

The code should look like the following code after those two changes:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>  
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch  
    {  
        (true, TimeBand.MorningRush, true)  => 2.00m,  
        (true, TimeBand.MorningRush, false) => 1.00m,  
        (true, TimeBand.Daytime,      _)    => 1.50m,  
        (true, TimeBand.EveningRush, true)  => 1.00m,  
        (true, TimeBand.EveningRush, false) => 2.00m,  
        (true, TimeBand.Overnight,      _)  => 0.75m,  
        (false, _,                  _)    => 1.00m,  
    };
```

Finally, you can remove the two rush hour times that pay the regular price. Once you remove those arms, you can replace the `false` with a discard (`_`) in the final switch arm. You'll have the following finished method:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>  
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch  
    {  
        (true, TimeBand.Overnight, _) => 0.75m,  
        (true, TimeBand.Daytime, _)  => 1.5m,  
        (true, TimeBand.MorningRush, true)  => 2.0m,  
        (true, TimeBand.EveningRush, false) => 2.0m,  
        _  => 1.0m,  
    };
```

This example highlights one of the advantages of pattern matching: the pattern branches are evaluated in order. If you rearrange them so that an earlier branch handles one of your later cases, the compiler warns you about the unreachable code. Those language rules made it easier to do the preceding simplifications with confidence that the code didn't change.

Pattern matching makes some types of code more readable and offers an alternative to object-oriented techniques when you can't add code to your classes. The cloud is causing data and functionality to live apart. The *shape* of the data and the *operations* on it aren't necessarily described together. In this tutorial, you consumed existing data in entirely different ways from its original function. Pattern matching gave you the ability to write functionality that overrode those types, even though you couldn't extend them.

Next steps

You can download the finished code from the [dotnet/samples](#) GitHub repository. Explore patterns on your own and add this technique into your regular coding activities. Learning these techniques gives you another way to approach problems and create new functionality.

Types (C# Programming Guide)

3/6/2021 • 12 minutes to read • [Edit Online](#)

Types, variables, and values

C# is a strongly typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method declaration specifies a name, number of parameters, and type and kind (value, reference, or output) for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types and more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library and user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following items:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The interface(s) it implements.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure all operations that are performed in your code are *type safe*. For example, if you declare a variable of type `int`, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type `bool`, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

NOTE

C and C++ developers, notice that in C#, `bool` is not convertible to `int`.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying types in variable declarations

When you declare a variable or constant in a program, you must either specify its type or use the `var` keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:  
float temperature;  
string name;  
MyClass myClass;  
  
// Declaration with initializers (four examples):  
char firstLetter = 'C';  
var limit = 3;  
int[] source = { 0, 1, 2, 3, 4, 5 };  
var query = from item in source  
            where item <= limit  
            select item;
```

The types of method parameters and return values are specified in the method declaration. The following signature shows a method that requires an [int](#) as an input argument and returns a string:

```
public string GetName(int ID)  
{  
    if (ID < names.Length)  
        return names[ID];  
    else  
        return String.Empty;  
}  
private string[] names = { "Spencer", "Sally", "Doug" };
```

After you declare a variable, you can't redeclare it with a new type, and you can't assign a value not compatible with its declared type. For example, you can't declare an [int](#) and then assign it a Boolean value of [true](#). However, values can be converted to other types, for example when they're assigned to new variables or passed as method arguments. A *type conversion* that doesn't cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see [Casting and Type Conversions](#).

Built-in types

C# provides a standard set of built-in types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in [string](#) and [object](#) types. These types are available for you to use in any C# program. For the complete list of the built-in types, see [Built-in types](#).

Custom types

You use the [struct](#), [class](#), [interface](#), [enum](#), and [record](#) constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they're defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see [.NET Class Library](#).

The common type system

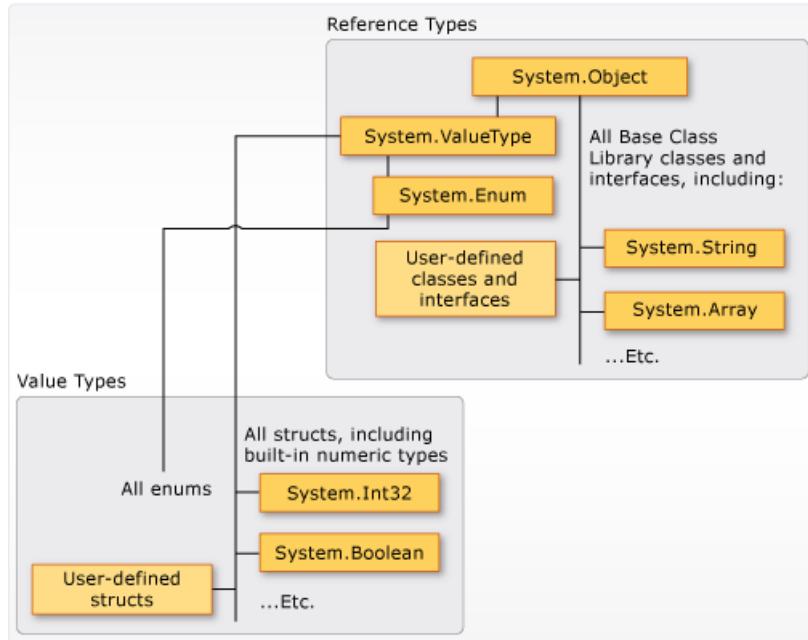
It's important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as [System.Int32](#) (C#

keyword: `int`), derive ultimately from a single base type, which is `System.Object` (C# keyword: `object`). This unified type hierarchy is called the [Common Type System](#) (CTS). For more information about inheritance in C#, see [Inheritance](#).

- Each type in the CTS is defined as either a *value type* or a *reference type*. These types include all custom types in the .NET class library and also your own user-defined types. Types that you define by using the `struct` keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the `class` or `record` keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



NOTE

You can see that the most commonly used types are all organized in the `System` namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value types

Value types derive from `System.ValueType`, which derives from `System.Object`. Types that derive from `System.ValueType` have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There's no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: `struct` and `enum`.

The built-in numeric types are structs, and they have fields and methods that you can access:

```
// constant field on type byte.  
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they're simple non-aggregate types:

```
byte num = 0xA;  
int i = 5;  
char c = 'Z';
```

Value types are *sealed*, which means that you can't derive a type from any value type, for example `System.Int32`.

You can't define a struct to inherit from any user-defined class or struct because a struct can only inherit from [System.ValueType](#). However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this cast causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a [System.Object](#) or any interface type as an input parameter. For more information, see [Boxing and Unboxing](#).

You use the [struct](#) keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

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

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see [Structure types](#). For more information about value types, see [Value types](#).

The other category of value types is [enum](#). An enum defines a set of named integral constants. For example, the [System.IO.FileMode](#) enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It's defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it's better to use enumerations instead of constant literal numbers. For more information, see [System.IO.FileMode](#).

All enums inherit from [System.Enum](#), which inherits from [System.ValueType](#). All the rules that apply to structs also apply to enums. For more information about enums, see [Enumeration types](#).

Reference types

A type that is defined as a [class](#), [record](#), [delegate](#), array, or [interface](#) is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value `null` until you explicitly create an object by using the `new` operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they're allocated and when they're reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it doesn't create a performance issue. For more information about garbage collection, see [Automatic Memory Management](#).

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the [System.Array](#) class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.  
int[] nums = { 1, 2, 3, 4, 5 };  
  
// Access an instance property of System.Array.  
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that isn't defined as [sealed](#), and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see [Classes and Structs](#). For more information about inheritance and virtual methods, see [Inheritance](#).

Types of literal values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see [Integral numeric types](#) and [Floating-point numeric types](#).

Because literals are typed, and all types derive ultimately from [System.Object](#), you can write and compile code such as the following code:

```
string s = "The answer is " + 5.ToString();  
// Outputs: "The answer is 5"  
Console.WriteLine(s);  
  
Type type = 12345.GetType();  
// Outputs: "System.Int32"  
Console.WriteLine(type);
```

Generic types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type [System.Collections.Generic.List<T>](#) has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to `object`. Generic collection classes are called *strongly typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see [Generics](#).

Implicit types, anonymous types, and nullable value types

As stated previously, you can implicitly type a local variable (but not class members) by using the `var` keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see [Implicitly Typed Local Variables](#).

It can be inconvenient to create a named type for simple sets of related values that you don't intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see [Anonymous Types](#).

Ordinary value types can't have a value of `null`. However, you can create nullable value types by appending a `?` after the type. For example, `int?` is an `int` type that can also have the value `null`. Nullable value types are instances of the generic struct type `System.Nullable<T>`. Nullable value types are especially useful when you're passing data to and from databases in which numeric values might be null. For more information, see [Nullable value types](#).

Compile-time type and runtime type

A variable can have different compile-time and run-time types. The *compile-time type* is the declared or inferred type of the variable in the source code. The *run-time type* is the type of the instance referred to by that variable. Often those two types are the same, as in the following example:

```
string message = "This is a string of characters";
```

In other cases, the compile-time type is different, as shown in the following two examples:

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

In both of the preceding examples, the run-time type is a `string`. The compile-time type is `object` in the first line, and `IEnumerable<char>` in the second.

If the two types are different for a variable, it's important to understand when the compile-time type and the run-time type apply. The compile-time type determines all the actions taken by the compiler. These compiler actions include method call resolution, overload resolution, and available implicit and explicit casts. The run-time type determines all actions that are resolved at run time. These run-time actions include dispatching virtual method calls, evaluating `is` and `switch` expressions, and other type testing APIs. To better understand how your code interacts with types, recognize which action applies to which type.

Related sections

For more information, see the following articles:

- [Casting and Type Conversions](#)
- [Boxing and Unboxing](#)
- [Using Type dynamic](#)
- [Value Types](#)
- [Reference Types](#)
- [Classes and Structs](#)
- [Anonymous Types](#)
- [Generics](#)

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Conversion of XML Data Types](#)
- [Integral types](#)

Nullable reference types

3/27/2021 • 9 minutes to read • [Edit Online](#)

C# 8.0 introduces **nullable reference types** and **non-nullable reference types** that enable you to make important statements about the properties for reference type variables:

- **A reference isn't supposed to be null.** When variables aren't supposed to be null, the compiler enforces rules that ensure it's safe to dereference these variables without first checking that it isn't null:
 - The variable must be initialized to a non-null value.
 - The variable can never be assigned the value `null`.
- **A reference may be null.** When variables may be null, the compiler enforces different rules to ensure that you've correctly checked for a null reference:
 - The variable may only be dereferenced when the compiler can guarantee that the value isn't null.
 - These variables may be initialized with the default `null` value and may be assigned the value `null` in other code.

This new feature provides significant benefits over the handling of reference variables in earlier versions of C# where the design intent can't be determined from the variable declaration. The compiler didn't provide safety against null reference exceptions for reference types:

- **A reference can be null.** The compiler doesn't issue warnings when a reference-type variable is initialized to `null`, or later assigned `null`. The compiler issues warnings when these variables are dereferenced without null checks.
- **A reference is assumed to be not null.** The compiler doesn't issue any warnings when reference types are dereferenced. The compiler issues warnings if a variable is set to an expression that may be null.

These warnings are emitted at compile time. The compiler doesn't add any null checks or other runtime constructs in a nullable context. At runtime, a nullable reference and a non-nullable reference are equivalent.

With the addition of nullable reference types, you can declare your intent more clearly. The `null` value is the correct way to represent that a variable doesn't refer to a value. Don't use this feature to remove all `null` values from your code. Rather, you should declare your intent to the compiler and other developers that read your code. By declaring your intent, the compiler informs you when you write code that is inconsistent with that intent.

A **nullable reference type** is noted using the same syntax as **nullable value types**: a `?` is appended to the type of the variable. For example, the following variable declaration represents a nullable string variable, `name`:

```
string? name;
```

Any variable where the `?` isn't appended to the type name is a **non-nullable reference type**. That includes all reference type variables in existing code when you've enabled this feature.

The compiler uses static analysis to determine if a nullable reference is known to be non-null. The compiler warns you if you dereference a nullable reference when it may be null. You can override this behavior by using the **null-forgiving operator** `!` following a variable name. For example, if you know the `name` variable isn't null but the compiler issues a warning, you can write the following code to override the compiler's analysis:

```
name!.Length;
```

Nullability of types

Any reference type can have one of four *nullabilities*, which describes when warnings are generated:

- *Nonnullable*: Null can't be assigned to variables of this type. Variables of this type don't need to be null-checked before dereferencing.
- *Nullable*: Null can be assigned to variables of this type. Dereferencing variables of this type without first checking for `null` causes a warning.
- *Oblivious*: Oblivious is the pre-C# 8.0 state. Variables of this type can be dereferenced or assigned without warnings.
- *Unknown*: Unknown is generally for type parameters where constraints don't tell the compiler that the type must be *nullable* or *nonnullable*.

The nullability of a type in a variable declaration is controlled by the *nullable context* in which the variable is declared.

Nullable contexts

Nullable contexts enable fine-grained control for how the compiler interprets reference type variables. The **nullable annotation context** of any given source line is either enabled or disabled. You can think of the pre-C# 8.0 compiler as compiling all your code in a disabled nullable context: any reference type may be null. The **nullable warnings context** may also be enabled or disabled. The nullable warnings context specifies the warnings generated by the compiler using its flow analysis.

The nullable annotation context and nullable warning context can be set for a project using the `Nullable` element in your `.csproj` file. This element configures how the compiler interprets the nullability of types and what warnings are generated. Valid settings are:

- `enable` : The nullable annotation context is **enabled**. The nullable warning context is **enabled**.
 - Variables of a reference type, `string` for example, are non-nullable. All nullability warnings are enabled.
- `warnings` : The nullable annotation context is **disabled**. The nullable warning context is **enabled**.
 - Variables of a reference type are oblivious. All nullability warnings are enabled.
- `annotations` : The nullable annotation context is **enabled**. The nullable warning context is **disabled**.
 - Variables of a reference type, `string` for example, are non-nullable. All nullability warnings are disabled.
- `disable` : The nullable annotation context is **disabled**. The nullable warning context is **disabled**.
 - Variables of a reference type are oblivious, just like earlier versions of C#. All nullability warnings are disabled.

Example:

```
<Nullable>enable</Nullable>
```

You can also use directives to set these same contexts anywhere in your project:

- `#nullable enable` : Sets the nullable annotation context and nullable warning context to **enabled**.
- `#nullable disable` : Sets the nullable annotation context and nullable warning context to **disabled**.
- `#nullable restore` : Restores the nullable annotation context and nullable warning context to the project settings.
- `#nullable disable warnings` : Set the nullable warning context to **disabled**.
- `#nullable enable warnings` : Set the nullable warning context to **enabled**.
- `#nullable restore warnings` : Restores the nullable warning context to the project settings.

- `#nullable disable annotations` : Set the nullable annotation context to **disabled**.
- `#nullable enable annotations` : Set the nullable annotation context to **enabled**.
- `#nullable restore annotations` : Restores the annotation warning context to the project settings.

IMPORTANT

The global nullable context does not apply for generated code files. Under either strategy, the nullable context is *disabled* for any source file marked as generated. This means any APIs in generated files are not annotated. There are four ways a file is marked as generated:

1. In the `.editorconfig`, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with `TemporaryGeneratedFile_`
4. End the file name with `.designer.cs`, `.generated.cs`, `.g.cs`, or `.g.i.cs`.

Generators can opt-in using the `#nullable` preprocessor directive.

By default, nullable annotation and warning contexts are **disabled**, including new projects. That means that your existing code compiles without changes and without generating any new warnings.

These options provide two distinct strategies to [update an existing codebase](#) to use nullable reference types.

Nullable annotation context

The compiler uses the following rules in a disabled nullable annotation context:

- You can't declare nullable references in a disabled context.
- All reference variables may be assigned a value of null.
- No warnings are generated when a variable of a reference type is dereferenced.
- The null-forgiving operator may not be used in a disabled context.

The behavior is the same as previous versions of C#.

The compiler uses the following rules in an enabled nullable annotation context:

- Any variable of a reference type is a **non-nullable reference**.
- Any non-nullable reference may be dereferenced safely.
- Any nullable reference type (noted by `?` after the type in the variable declaration) may be null. Static analysis determines if the value is known to be non-null when it's dereferenced. If not, the compiler warns you.
- You can use the null-forgiving operator to declare that a nullable reference isn't null.

In an enabled nullable annotation context, the `?` character appended to a reference type declares a **nullable reference type**. The **null-forgiving operator** `!` may be appended to an expression to declare that the expression isn't null.

Nullable warning context

The nullable warning context is distinct from the nullable annotation context. Warnings can be enabled even when the new annotations are disabled. The compiler uses static flow analysis to determine the **null state** of any reference. The null state is either **not null** or **maybe null** when the *nullable warning context* isn't **disabled**. If you dereference a reference when the compiler has determined it's **maybe null**, the compiler warns you. The state of a reference is **maybe null** unless the compiler can determine one of two conditions:

1. The variable has been definitely assigned a non-null value.
2. The variable or expression has been checked against null before de-referencing it.

The compiler generates warnings when you dereference a variable or expression that is **maybe null** in a nullable warning context. Furthermore, the compiler generates warnings when a nonnullable reference-type variable is assigned a **maybe null** variable or expression in an enabled nullable annotation context.

Attributes describe APIs

You add attributes to APIs that provide the compiler more information about when arguments or return values can or can't be null. You can learn more about these attributes in our article in the language reference covering the [nullable attributes](#). These attributes are being added to .NET libraries over current and upcoming releases. The most commonly used APIs are being updated first.

Known pitfalls

Arrays and structs that contain reference types are known pitfalls in nullable reference types feature.

Structs

A struct that contains non-nullable reference types allows assigning `default` for it without any warnings. Consider the following example:

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

In the preceding example, there is no warning in `PrintStudent(default)` while the non-nullable reference types `FirstName` and `LastName` are null.

Another more common case is when you deal with generic structs. Consider the following example:

```
#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}
```

In the preceding example, the property `Bar` is going to be `null` at runtime, and it's assigned to non-nullable string without any warnings.

Arrays

Arrays are also a known pitfall in nullable reference types. Consider the following example which doesn't produce any warnings:

```
using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

In the preceding example, the declaration of the array shows it holds non-nullable strings, while its elements are all initialized to null. Then, the variable `s` is assigned a null value (the first element of the array). Finally, the variable `s` is dereferenced causing a runtime exception.

See also

- [Draft nullable reference types specification](#)
- [Intro to nullable references tutorial](#)
- [Migrate an existing codebase to nullable references](#)
- [Nullable \(C# Compiler option\)](#)

Update libraries to use nullable reference types and communicate nullable rules to callers

3/27/2021 • 10 minutes to read • [Edit Online](#)

The addition of [nullable reference types](#) means you can declare whether or not a `null` value is allowed or expected for every variable. In addition, you can apply a number of attributes: `AllowNull`, `DisallowNull`, `MaybeNull`, `NotNull`, `NotNullWhen`, `MaybeNullWhen`, and `NotNullIfNotNull` to completely describe the null states of argument and return values. That provides a great experience as you write code. You get warnings if a non-nullable variable might be set to `null`. You get warnings if a nullable variable isn't null-checked before you dereference it. Updating your libraries can take time, but the payoffs are worth it. The more information you provide to the compiler about *when* a `null` value is allowed or prohibited, the better warnings users of your API will get. Let's start with a familiar example. Imagine your library has the following API to retrieve a resource string:

```
bool TryGetMessage(string key, out string message)
```

The preceding example follows the familiar `Try*` pattern in .NET. There are two reference arguments for this API: the `key` and the `message` parameter. This API has the following rules relating to the nullness of these arguments:

- Callers shouldn't pass `null` as the argument for `key`.
- Callers can pass a variable whose value is `null` as the argument for `message`.
- If the `TryGetMessage` method returns `true`, the value of `message` isn't null. If the return value is `false`, the value of `message` (and its null state) is null.

The rule for `key` can be completely expressed by the variable type: `key` should be a non-nullable reference type. The `message` parameter is more complex. It allows `null` as the argument, but guarantees that, on success, that `out` argument isn't null. For these scenarios, you need a richer vocabulary to describe the expectations.

Updating your library for nullable references requires more than sprinkling `?` on some of the variables and type names. The preceding example shows that you need to examine your APIs and consider your expectations for each input argument. Consider the guarantees for the return value, and any `out` or `ref` arguments upon the method's return. Then communicate those rules to the compiler, and the compiler will provide warnings when callers don't abide by those rules.

This work takes time. Let's start with strategies to make your library or application nullable-aware, while balancing other requirements. You'll see how to balance ongoing development enabling nullable reference types. You'll learn challenges for generic type definitions. You'll learn to apply attributes to describe pre- and post-conditions on individual APIs.

Choose a strategy for nullable reference types

The first choice is whether nullable reference types should be on or off by default. You have two strategies:

- Enable nullable reference types for the entire project, and disable it in code that's not ready.
- Only enable nullable reference types for code that's been annotated for nullable reference types.

The first strategy works best when you're adding other features to the library as you update it for nullable reference types. All new development is nullable aware. As you update existing code, you enable nullable

reference types in those classes.

Following this first strategy, you do the following steps:

1. Enable nullable reference types for the entire project by adding the `<Nullable>enable</Nullable>` element to your `csproj` files.
2. Add the `#nullable disable` pragma to every source file in your project.
3. As you work on each file, remove the pragma and address any warnings.

This first strategy has more up-front work to add the pragma to every file. The advantage is that every new code file added to the project will be nullable enabled. Any new work will be nullable aware; only existing code must be updated.

The second strategy works better if the library is stable, and the main focus of the development is to adopt nullable reference types. You turn on nullable reference types as you annotate APIs. When you've finished, you enable nullable reference types for the entire project.

Following this second strategy you do the following steps:

1. Add the `#nullable enable` pragma to the file you want to make nullable aware.
2. Address any warnings.
3. Continue these first two steps until you've made the entire library nullable aware.
4. Enable nullable types for the entire project by adding the `<Nullable>enable</Nullable>` element to your `csproj` files.
5. Remove the `#nullable enable` pragmas, as they're no longer needed.

This second strategy has less work up-front. The tradeoff is that the first task when you create a new file is to add the pragma and make it nullable aware. If any developers on your team forget, that new code is now in the backlog of work to make all code nullable aware.

Which of these strategies you pick depends on how much active development is taking place in your project. The more mature and stable your project, the better the second strategy. The more features being developed, the better the first strategy.

IMPORTANT

The global nullable context does not apply for generated code files. Under either strategy, the nullable context is *disabled* for any source file marked as generated. This means any APIs in generated files are not annotated. There are four ways a file is marked as generated:

1. In the `.editorconfig`, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with `TemporaryGeneratedFile_`
4. End the file name with `.designer.cs`, `.generated.cs`, `.g.cs`, or `.g.i.cs`.

Generators can opt-in using the `#nullable` preprocessor directive.

Should nullable warnings introduce breaking changes?

Before you enable nullable reference types, variables are considered *nullable oblivious*. Once you enable nullable reference types, all those variables are *non-nullable*. The compiler will issue warnings if those variables aren't initialized to non-null values.

Another likely source of warnings is return values when the value hasn't been initialized.

The first step in addressing the compiler warnings is to use `? annotations` on parameter and return types to indicate when arguments or return values may be null. When reference variables must not be null, the original declaration is correct. As you do this task, your goal isn't just to fix warnings. The more important goal is to make the compiler understand your intent for potential null values. As you examine the warnings, you reach your next major decision for your library. Do you want to consider modifying API signatures to more clearly communicate your design intent? A better API signature for the `TryGetMessage` method examined earlier could be:

```
string? TryGetMessage(string key);
```

The return value indicates success or failure, and carries the value if the value was found. In many cases, changing API signatures can improve how they communicate null values.

However, for public libraries, or libraries with large user bases, you may prefer not introducing any API signature changes. For those cases, and other common patterns, you can apply attributes to more clearly define when an argument or return value may be `null`. Whether or not you consider changing the surface of your API, you'll likely find that type annotations alone aren't sufficient for describing `null` values for arguments or return values. In those instances, you can apply attributes to more clearly describe an API.

Attributes extend type annotations

Several attributes have been added to express additional information about the null state of variables. All code you wrote before C# 8 introduced nullable reference types was *null oblivious*. That means any reference type variable may be null, but null checks aren't required. Once your code is *nullable aware*, those rules change. Reference types should never be the `null` value, and nullable reference types must be checked against `null` before being dereferenced.

The rules for your APIs are likely more complicated, as you saw with the `TryGetValue` API scenario. Many of your APIs have more complex rules for when variables can or can't be `null`. In these cases, you'll use attributes to express those rules. The attributes that describe the semantics of your API are found in the article on [Attributes that impact nullable analysis](#).

Generic definitions and nullability

Correctly communicating the null state of generic types and generic methods requires special care. The extra care stems from the fact that a nullable value type and a nullable reference type are fundamentally different. An `int?` is a synonym for `Nullable<int>`, whereas `string?` is `string` with an attribute added by the compiler. The result is that the compiler can't generate correct code for `T?` without knowing if `T` is a `class` or a `struct`.

This fact doesn't mean you can't use a nullable type (either value type or reference type) as the type argument for a closed generic type. Both `List<string?>` and `List<int?>` are valid instantiations of `List<T>`.

What it does mean is that you can't use `T?` in a generic class or method declaration without constraints. For example, `Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>)` won't be changed to return `T?`. You can overcome this limitation by adding either the `struct` or `class` constraint. With either of those constraints, the compiler knows how to generate code for both `T` and `T?`.

You may want to restrict the types used for a generic type argument to be non-nullable types. You can do that by adding the `notnull` constraint on that type argument. When that constraint is applied, the type argument must not be a nullable type.

Late-initialized properties, Data Transfer Objects, and nullability

Indicating the nullability of properties that are late-initialized, meaning set after construction, may require special consideration to ensure that your class continues to correctly express the original design intent.

Types that contain late-initialized properties, such as Data Transfer Objects (DTOs), are often instantiated by an external library, like a database ORM (Object Relational Mapper), a deserializer, or some other component that automatically populates properties from another source.

Consider the following DTO class, prior to enabling nullable reference types, that represents a student:

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string VehicleRegistration { get; set; }
}
```

The design intent (indicated in this case by the `[Required]` attribute) suggests that in this system, the `FirstName` and `LastName` properties are **mandatory**, and therefore not null.

The `VehicleRegistration` property is **not mandatory**, so may be null.

When you enable nullable reference types, you want to indicate which properties on your DTO may be nullable, consistent with your original intent:

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}
```

For this DTO, the only property that may be null is `VehicleRegistration`.

However, the compiler raises `CS8618` warnings for both `FirstName` and `LastName`, indicating the non-nullable properties are uninitialized.

There are three options available to you that resolve the compiler warnings in a way that maintains the original intent. Any of these options are valid; you should choose the one that best suits your coding style and design requirements.

Initialize in the constructor

The ideal way to resolve the uninitialized warnings is to initialize the properties in the constructor:

```

class Student
{
    public Student(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}

```

This approach only works if the library that you use to instantiate the class supports passing parameters in the constructor.

A library may support passing *some* properties in the constructor, but not all. For example, EF Core supports [constructor binding](#) for normal column properties, but not navigation properties.

Check the documentation on the library that instantiates your class, to understand the extent to which it supports constructor binding.

Property with nullable backing field

If constructor binding won't work for you, one way to deal with this problem is to have a non-nullable property with a nullable backing field:

```

private string? _firstName;

[Required]
public string FirstName
{
    set => _firstName = value;
    get => _firstName
        ?? throw new InvalidOperationException("Uninitialized " + nameof(FirstName))
}

```

In this scenario, if the `FirstName` property is accessed before it has been initialized, then the code throws an `InvalidOperationException`, because the API contract has been used incorrectly.

Consider that some libraries may have special considerations when using backing fields. For example, EF Core may need to be configured to use [backing fields](#) correctly.

Initialize the property to null

As a terser alternative to using a nullable backing field, or if the library that instantiates your class isn't compatible with that approach, you can initialize the property to `null` directly, with the help of the null-forgiving operator (`!`):

```

[Required]
public string FirstName { get; set; } = null!;

[Required]
public string LastName { get; set; } = null!;

public string? VehicleRegistration { get; set; }

```

You'll never observe an actual null value at runtime except as a result of a programming bug, by accessing the property before it has been properly initialized.

See also

- [Migrate an existing codebase to nullable references](#)
- [Working with Nullable Reference Types in EF Core](#)

Namespaces (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Namespaces are heavily used in C# programming in two ways. First, .NET uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name is not required, as in the following example:

```
using System;
```

```
Console.WriteLine("Hello World!");
```

For more information, see the [using Directive](#).

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the `namespace` keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# [identifier name](#).

Namespaces overview

Namespaces have the following properties:

- They organize large code projects.
- They are delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET `System` namespace.

C# language specification

For more information, see the [Namespaces](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Using Namespaces](#)
- [How to use the My namespace](#)
- [Identifier names](#)
- [using Directive](#)
- [:: Operator](#)

Types, variables, and values

3/6/2021 • 6 minutes to read • [Edit Online](#)

C# is a strongly typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The interface(s) it implements.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type `int`, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type `bool`, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

NOTE

C and C++ developers, notice that in C#, `bool` is not convertible to `int`.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying types in variable declarations

When you declare a variable or constant in a program, you must either specify its type or use the `var` keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:  
float temperature;  
string name;  
MyClass myClass;  
  
// Declaration with initializers (four examples):  
char firstLetter = 'C';  
var limit = 3;  
int[] source = { 0, 1, 2, 3, 4, 5 };  
var query = from item in source  
            where item <= limit  
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an [int](#) as an input argument and returns a string:

```
public string GetName(int ID)  
{  
    if (ID < names.Length)  
        return names[ID];  
    else  
        return String.Empty;  
}  
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an [int](#) and then assign it a Boolean value of [true](#). However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see [Casting and type conversions](#).

Built-in types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in [string](#) and [object](#) types. These are available for you to use in any C# program. For the complete list of the built-in types, see [Built-in types](#).

Custom types

You use the [struct](#), [class](#), [record](#), [interface](#), and [enum](#) constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code.

Generic types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, [List<T>](#) has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, [string](#):

```
List<string> strings = new List<string>();
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to [object](#). Generic collection classes are called *strongly typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `strings` object in the previous example. For more information, see [Generics](#).

Implicit types, anonymous types, and tuple types

As stated previously, you can implicitly type a local variable (but not class members) by using the [var](#) keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see [Implicitly typed local variables](#).

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see [Anonymous types](#).

It's common to want to return more than one value from a method. You can create *tuple types* that return multiple values in a single method call. For more information, see [Tuple types](#).

The Common type system

It is important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as [Int32](#) (C# keyword: `int`), derive ultimately from a single base type, which is [Object](#) (C# keyword: `object`). This unified type hierarchy is called the [Common type system](#) (CTS). For more information about inheritance in C#, see [Inheritance](#).
- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET class library and also your own user-defined types. Types that you define by using the `struct` or `enum` keyword are value types. For more information about value types, see [Value types](#). Types that you define by using the `class` keyword are reference types. For more information about reference types, see [Classes](#). Reference types and value types have different compile-time rules, and different run-time behavior.

See also

- [Structure types](#)
- [Enumeration types](#)
- [Classes](#)

Classes (C# Programming Guide)

3/6/2021 • 5 minutes to read • [Edit Online](#)

Reference types

A type that is defined as a [class](#) is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value [null](#) until you explicitly create an instance of the class by using the [new](#) operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

```
//Declaring an object of type MyClass.  
MyClass mc = new MyClass();  
  
//Declaring another object of the same type, assigning it the value of the first object.  
MyClass mc2 = mc;
```

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized and in most scenarios, it does not create a performance issue. For more information about garbage collection, see [Automatic memory management and garbage collection](#).

Declaring Classes

Classes are declared by using the [class](#) keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]  
public class Customer  
{  
    // Fields, properties, methods and events go here...  
}
```

The [class](#) keyword is preceded by the access level. Because [public](#) is used in this case, anyone can create instances of this class. The name of the class follows the [class](#) keyword. The name of the class must be a valid C# [identifier name](#). The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the [new](#) keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous example, `object1` is a reference to an object that is based on `Customer`. This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

We don't recommend creating object references such as this one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it an existing object, such as this:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through `object3` are reflected in subsequent uses of `object4`. Because objects that are based on classes are referred to by reference, classes are known as reference types.

Class inheritance

Classes fully support *inheritance*, a fundamental characteristic of object-oriented programming. When you create a class, you can inherit from any other class that is not defined as `sealed`, and other classes can inherit from your class and override class virtual methods. Furthermore, you can implement one or more interfaces.

Inheritance is accomplished by using a *derivation*, which means a class is declared by using a *base class* from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see [Inheritance](#).

Unlike C++, a class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement one or more interfaces. For more information, see [Interfaces](#).

A class can be declared `abstract`. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a `sealed` class does not allow other classes to derive from it. For more information, see [Abstract and Sealed Classes and Class Members](#).

Class definitions can be split between different source files. For more information, see [Partial Classes and Methods](#).

Example

The following example defines a public class that contains an [auto-implemented property](#), a method, and a special method called a constructor. For more information, see [Properties, Methods](#), and [Constructors](#) topics. The instances of the class are then instantiated with the `new` keyword.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Object-Oriented Programming](#)
- [Polymorphism](#)
- [Identifier names](#)
- [Members](#)
- [Methods](#)

- Constructors
- Finalizers
- Objects

Deconstructing tuples and other types

3/23/2021 • 9 minutes to read • [Edit Online](#)

A tuple provides a lightweight way to retrieve multiple values from a method call. But once you retrieve the tuple, you have to handle its individual elements. Doing this on an element-by-element basis is cumbersome, as the following example shows. The `QueryCityData` method returns a 3-tuple, and each of its elements is assigned to a variable in a separate operation.

```
using System;

public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

        // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Retrieving multiple field and property values from an object can be equally cumbersome: you have to assign a field or property value to a variable on a member-by-member basis.

Starting with C# 7.0, you can retrieve multiple elements from a tuple or retrieve multiple field, property, and computed values from an object in a single *deconstruct* operation. When you deconstruct a tuple, you assign its elements to individual variables. When you deconstruct an object, you assign selected values to individual variables.

Deconstructing a tuple

C# features built-in support for deconstructing tuples, which lets you unpack all the items in a tuple in a single operation. The general syntax for deconstructing a tuple is similar to the syntax for defining one: you enclose the variables to which each element is to be assigned in parentheses in the left side of an assignment statement. For example, the following statement assigns the elements of a 4-tuple to four separate variables:

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

There are three ways to deconstruct a tuple:

- You can explicitly declare the type of each field inside parentheses. The following example uses this approach to deconstruct the 3-tuple returned by the `QueryCityData` method.

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- You can use the `var` keyword so that C# infers the type of each variable. You place the `var` keyword outside of the parentheses. The following example uses type inference when deconstructing the 3-tuple returned by the `QueryCityData` method.

```
public static void Main()
{
    var (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

You can also use the `var` keyword individually with any or all of the variable declarations inside the parentheses.

```
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York City");

    // Do something with the data.
}
```

This is cumbersome and is not recommended.

- Lastly, you may deconstruct the tuple into variables that have already been declared.

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Note that you cannot specify a specific type outside the parentheses even if every field in the tuple has the same type. This generates compiler error CS8136, "Deconstruction 'var (...)' form disallows a specific type for 'var'."

Note that you must also assign each element of the tuple to a variable. If you omit any elements, the compiler generates error CS8132, "Cannot deconstruct a tuple of 'x' elements into 'y' variables."

Note that you cannot mix declarations and assignments to existing variables on the left-hand side of a deconstruction. The compiler generates error CS8184, "a deconstruction cannot mix declarations and expressions on the left-hand-side." when the members include newly declared and existing variables.

Deconstructing tuple elements with discards

Often when deconstructing a tuple, you're interested in the values of only some elements. Starting with C# 7.0, you can take advantage of C#'s support for *discards*, which are write-only variables whose values you've chosen

to ignore. A discard is designated by an underscore character ("_") in an assignment. You can discard as many values as you like; all are represented by the single discard, `_`.

The following example illustrates the use of tuples with discards. The `QueryCityDataForYears` method returns a 6-tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}

// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Deconstructing user-defined types

C# does not offer built-in support for deconstructing non-tuple types other than the `record` and `DictionaryEntry` types. However, as the author of a class, a struct, or an interface, you can allow instances of the type to be deconstructed by implementing one or more `Deconstruct` methods. The method returns void, and each value to be deconstructed is indicated by an `out` parameter in the method signature. For example, the following `Deconstruct` method of a `Person` class returns the first, middle, and last name:

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

You can then deconstruct an instance of the `Person` class named `p` with an assignment like the following:

```
var (fName, mName, lName) = p;
```

The following example overloads the `Deconstruct` method to return various combinations of properties of a `Person` object. Individual overloads return:

- A first and last name.
- A first, middle, and last name.
- A first name, a last name, a city name, and a state name.

```

using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                           out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class Example
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//     Hello John Adams of Boston, MA!

```

Multiple `Deconstruct` methods having the same number of parameters are ambiguous. You must be careful to define `Deconstruct` methods with different numbers of parameters, or "arity". `Deconstruct` methods with the same number of parameters cannot be distinguished during overload resolution.

Deconstructing a user-defined type with discards

Just as you do with [tuples](#), you can use discards to ignore selected items returned by a `Deconstruct` method.

Each discard is defined by a variable named `_`, and a single deconstruction operation can include multiple discards.

The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state) but discards the last name and the state.

```
// Deconstruct the person object.  
var (fName, _, city, _) = p;  
Console.WriteLine($"Hello {fName} of {city}!");  
// The example displays the following output:  
//     Hello John of Boston!
```

Deconstructing a user-defined type with an extension method

If you didn't author a class, struct, or interface, you can still deconstruct objects of that type by implementing one or more `Deconstruct` [extension methods](#) to return the values in which you're interested.

The following example defines two `Deconstruct` extension methods for the `System.Reflection.PropertyInfo` class. The first returns a set of values that indicate the characteristics of the property, including its type, whether it's static or instance, whether it's read-only, and whether it's indexed. The second indicates the property's accessibility. Because the accessibility of get and set accessors can differ, Boolean values indicate whether the property has separate get and set accessors and, if it does, whether they have the same accessibility. If there is only one accessor or both the get and the set accessor have the same accessibility, the `access` variable indicates the accessibility of the property as a whole. Otherwise, the accessibility of the get and set accessors are indicated by the `getAccess` and `setAccess` variables.

```
using System;  
using System.Collections.Generic;  
using System.Reflection;  
  
public static class ReflectionExtensions  
{  
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,  
                                  out bool isReadOnly, out bool isIndexed,  
                                  out Type propertyType)  
    {  
        var getter = p.GetMethod();  
  
        // Is the property read-only?  
        isReadOnly = !p.CanWrite;  
  
        // Is the property instance or static?  
        isStatic = getter.IsStatic;  
  
        // Is the property indexed?  
        isIndexed = p.GetIndexParameters().Length > 0;  
  
        // Get the property type.  
        propertyType = p.PropertyType;  
    }  
  
    public static void Deconstruct(this PropertyInfo p, out bool hasGetAndSet,  
                                  out bool sameAccess, out string access,  
                                  out string getAccess, out string setAccess)  
    {  
        hasGetAndSet = sameAccess = false;  
        string getAccessTemp = null;  
        string setAccessTemp = null;  
  
        MethodInfo getter = null;  
        if (p.CanRead)
```

```

getter = p.GetMethod();

MethodInfo setter = null;
if (p.CanWrite)
    setter = p.SetMethod();

if (setter != null && getter != null)
    hasGetAndSet = true;

if (getter != null)
{
    if (getter.IsPublic)
        getAccessTemp = "public";
    else if (getter.IsPrivate)
        getAccessTemp = "private";
    else if (getter.IsAssembly)
        getAccessTemp = "internal";
    else if (getter.IsFamily)
        getAccessTemp = "protected";
    else if (getter.IsFamilyOrAssembly)
        getAccessTemp = "protected internal";
}

if (setter != null)
{
    if (setter.IsPublic)
        setAccessTemp = "public";
    else if (setter.IsPrivate)
        setAccessTemp = "private";
    else if (setter.IsAssembly)
        setAccessTemp = "internal";
    else if (setter.IsFamily)
        setAccessTemp = "protected";
    else if (setter.IsFamilyOrAssembly)
        setAccessTemp = "protected internal";
}

// Are the accessibility of the getter and setter the same?
if (setAccessTemp == getAccessTemp)
{
    sameAccess = true;
    access = getAccessTemp;
    getAccess = setAccess = String.Empty;
}
else
{
    access = null;
    getAccess = getAccessTemp;
    setAccess = setAccessTemp;
}
}

public class Example
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\\nThe {dateType.FullName}.{prop.Name} property:");
        Console.WriteLine($"    PropertyType: {propType.Name}");
        Console.WriteLine($"    Static:      {isStatic}");
        Console.WriteLine($"    Read-only:   {isRO}");
        Console.WriteLine($"    Indexed:     {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |

```

```
BindingFlags.Static);
    var (hasGetAndSet, sameAccess, accessibility, getAccessibility, setAccessibility) = prop;
    Console.WriteLine($"\\nAccessibility of the {listType.FullName}.{prop.Name} property: ");

    if (!hasGetAndSet | sameAccess)
    {
        Console.WriteLine(accessibility);
    }
    else
    {
        Console.WriteLine($"\\n    The get accessor: {getAccessibility}");
        Console.WriteLine($"    The set accessor: {setAccessibility}");
    }
}

// The example displays the following output:
//     The System.DateTime.Now property:
//         PropertyType: DateTime
//         Static:      True
//         Read-only:   True
//         Indexed:    False
//
//     Accessibility of the System.Collections.Generic.List`1.Item property: public
```

Deconstructing a `record` type

When you declare a `record` type by using two or more positional parameters, the compiler creates a `Deconstruct` method with an `out` parameter for each positional parameter in the `record` declaration. For more information, see [Positional syntax for property definition](#) and [Deconstructor behavior in derived records](#).

See also

- [Discards](#)
- [Tuple types](#)

Interfaces (C# Programming Guide)

3/23/2021 • 4 minutes to read • [Edit Online](#)

An interface contains definitions for a group of related functionalities that a non-abstract [class](#) or a [struct](#) must implement. An interface may define [static](#) methods, which must have an implementation. Beginning with C# 8.0, an interface may define a default implementation for members. An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the [interface](#) keyword as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of an interface must be a valid C# [identifier name](#). By convention, interface names begin with a capital [I](#).

Any class or struct that implements the [IEquatable<T>](#) interface must contain a definition for an [Equals](#) method that matches the signature that the interface specifies. As a result, you can count on a class that implements [IEquatable<T>](#) to contain an [Equals](#) method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of [IEquatable<T>](#) doesn't provide an implementation for [Equals](#). A class or struct can implement multiple interfaces, but a class can only inherit from a single class.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members](#).

Interfaces can contain instance methods, properties, events, indexers, or any combination of those four member types. Interfaces may contain static constructors, fields, constants, or operators. For links to examples, see [Related Sections](#).

An interface can't contain instance fields, instance constructors, or finalizers. Interface members are public by default, and you can explicitly specify accessibility modifiers, such as [public](#), [protected](#), [internal](#), [private](#), [protected internal](#), or [private protected](#). A [private](#) member must have a default implementation.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface declares but doesn't provide a default implementation for. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the [IEquatable<T>](#) interface. The implementing class, [Car](#), must provide an implementation of the [Equals](#) method.

```

public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}

```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a `get` accessor. The class that implements the interface can declare the same property with both a `get` and `set` accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see [Explicit Interface Implementation](#) and [Interface Properties](#).

Interfaces can inherit from one or more interfaces. The derived interface inherits the members from its base interfaces. A class that implements a derived interface must implement all members in the derived interface, including all members of the derived interface's base interfaces. That class may be implicitly converted to the derived interface or any of its base interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation. When interfaces declare a default implementation of a method, any class implementing that interface inherits that implementation. Implementations defined in interfaces are virtual and the implementing class may override that implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see [Polymorphism](#).

Interfaces summary

An interface has the following properties:

- An interface is typically like an abstract base class with only abstract members. Any class or struct that implements the interface must implement all its members. Optionally, an interface may define default implementations for some or all of its members. For more information, see [default interface methods](#).
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

Related Sections

- [Interface Properties](#)
- [Indexers in Interfaces](#)
- [How to implement interface events](#)
- [Classes and Structs](#)

- [Inheritance](#)
- [Interfaces](#)
- [Methods](#)
- [Polymorphism](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Properties](#)
- [Events](#)
- [Indexers](#)

See also

- [C# Programming Guide](#)
- [Inheritance](#)
- [Identifier names](#)

Methods in (C#)

3/18/2021 • 21 minutes to read • [Edit Online](#)

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The `Main` method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

NOTE

This topic discusses named methods. For information about anonymous functions, see [Anonymous Functions](#).

Method signatures

Methods are declared in a `class` or `struct` by specifying:

- An optional access level, such as `public` or `private`. The default is `private`.
- Optional modifiers such as `abstract` or `sealed`.
- The return value, or `void` if the method has none.
- The method name.
- Any method parameters. Method parameters are enclosed in parentheses and are separated by commas.
Empty parentheses indicate that the method requires no parameters.

These parts together form the method signature.

IMPORTANT

A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

The following example defines a class named `Motorcycle` that contains five methods:

```

using System;

abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() { /* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes can override the base class implementation.
    public virtual int Drive(TimeSpan time, int speed) { /* Method statements here */ return 0; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}

```

Note that the `Motorcycle` class includes an overloaded method, `Drive`. Two methods have the same name, but must be differentiated by their parameter types.

Method invocation

Methods can be either *instance* or *static*. Invoking an instance method requires that you instantiate an object and call the method on that object; an instance method operates on that instance and its data. You invoke a static method by referencing the name of the type to which the method belongs; static methods do not operate on instance data. Attempting to call a static method through an object instance generates a compiler error.

Calling a method is like accessing a field. After the object name (if you are calling an instance method) or the type name (if you are calling a `static` method), add a period, the name of the method, and parentheses. Arguments are listed within the parentheses and are separated by commas.

The method definition specifies the names and types of any parameters that are required. When a caller invokes the method, it provides concrete values, called arguments, for each parameter. The arguments must be compatible with the parameter type, but the argument name, if one is used in the calling code, does not have to be the same as the parameter named defined in the method. In the following example, the `Square` method includes a single parameter of type `int` named `i`. The first method call passes the `Square` method a variable of type `int` named `num`; the second, a numeric constant; and the third, an expression.

```

public class Example
{
    public static void Main()
    {
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}

```

The most common form of method invocation used positional arguments; it supplies arguments in the same order as method parameters. The methods of the `Motorcycle` class can therefore be called as in the following example. The call to the `Drive` method, for example, includes two arguments that correspond to the two parameters in the method's syntax. The first becomes the value of the `miles` parameter, the second the value of the `speed` parameter.

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

You can also use *named arguments* instead of positional arguments when invoking a method. When using named arguments, you specify the parameter name followed by a colon (":") and the argument. Arguments to the method can appear in any order, as long as all required arguments are present. The following example uses named arguments to invoke the `TestMotorcycle.Drive` method. In this example, the named arguments are passed in the opposite order from the method's parameter list.

```

using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int) Math.Round( ((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}

// The example displays the following output:
//      Travel time: approx. 3 hours

```

You can invoke a method using both positional arguments and named arguments. However, positional arguments can only follow named arguments when the named arguments are in the correct positions. The following example invokes the `TestMotorcycle.Drive` method from the previous example using one positional argument and one named argument.

```
var travelTime = moto.Drive(170, speed: 55);
```

Inherited and overridden methods

In addition to the members that are explicitly defined in a type, a type inherits members defined in its base classes. Since all types in the managed type system inherit directly or indirectly from the `Object` class, all types inherit its members, such as `Equals(Object)`, `GetType()`, and `ToString()`. The following example defines a `Person` class, instantiates two `Person` objects, and calls the `Person.Equals` method to determine whether the two objects are equal. The `Equals` method, however, is not defined in the `Person` class; it is inherited from `Object`.

```

using System;

public class Person
{
    public String FirstName;
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: False

```

Types can override inherited members by using the `override` keyword and providing an implementation for the overridden method. The method signature must be the same as that of the overridden method. The following example is like the previous one, except that it overrides the [Equals\(Object\)](#) method. (It also overrides the [GetHashCode\(\)](#) method, since the two methods are intended to provide consistent results.)

```

using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: True

```

Passing parameters

Types in C# are either *value types* or *reference types*. For a list of built-in value types, see [Types](#). By default, both value types and reference types are passed to a method by value.

Passing parameters by value

When a value type is passed to a method by value, a copy of the object instead of the object itself is passed to the method. Therefore, changes to the object in the called method have no effect on the original object when control returns to the caller.

The following example passes a value type to a method by value, and the called method attempts to change the value type's value. It defines a variable of type `int`, which is a value type, initializes its value to 20, and passes it to a method named `ModifyValue` that changes the variable's value to 30. When the method returns, however, the variable's value remains unchanged.

```
using System;

public class Example
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

When an object of a reference type is passed to a method by value, a reference to the object is passed by value. That is, the method receives not the object itself, but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the object when control returns to the calling method. However, replacing the object passed to the method has no effect on the original object when control returns to the caller.

The following example defines a class (which is a reference type) named `SampleRefType`. It instantiates a `SampleRefType` object, assigns 44 to its `value` field, and passes the object to the `ModifyObject` method. This example does essentially the same thing as the previous example -- it passes an argument by value to a method. But because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `obj.value` field also changes the `value` field of the argument, `rt`, in the `Main` method to 33, as the output from the example shows.

```

using System;

public class SampleRefType
{
    public int value;
}

public class Example
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}

```

Passing parameters by reference

You pass a parameter by reference when you want to change the value of an argument in a method and want to reflect that change when control returns to the calling method. To pass a parameter by reference, you use the `ref` or `out` keyword. You can also pass a value by reference to avoid copying but still prevent modifications using the `in` keyword.

The following example is identical to the previous one, except the value is passed by reference to the `ModifyValue` method. When the value of the parameter is modified in the `ModifyValue` method, the change in value is reflected when control returns to the caller.

```

using System;

public class Example
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30

```

A common pattern that uses `ref` parameters involves swapping the values of variables. You pass two variables to a method by reference, and the method swaps their contents. The following example swaps integer values.

```

using System;

public class Example
{
    static void Main()
    {
        int i = 2, j = 3;
        System.Console.WriteLine("i = {0}  j = {1}" , i, j);

        Swap(ref i, ref j);

        System.Console.WriteLine("i = {0}  j = {1}" , i, j);
    }

    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2

```

Passing a reference-type parameter allows you to change the value of the reference itself, rather than the value of its individual elements or fields.

Parameter arrays

Sometimes, the requirement that you specify the exact number of arguments to your method is restrictive. By using the `params` keyword to indicate that a parameter is a parameter array, you allow your method to be called with a variable number of arguments. The parameter tagged with the `params` keyword must be an array type, and it must be the last parameter in the method's parameter list.

A caller can then invoke the method in either of four ways:

- By passing an array of the appropriate type that contains the desired number of elements.
- By passing a comma-separated list of individual arguments of the appropriate type to the method.
- By passing `null`.
- By not providing an argument to the parameter array.

The following example defines a method named `GetVowels` that returns all the vowels from a parameter array. The `Main` method illustrates all four ways of invoking the method. Callers are not required to supply any arguments for parameters that include the `params` modifier. In that case, the parameter is an empty array.

```

using System;
using System.Linq;

class Example
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments: '{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaaea'
//     Vowels from multiple arguments: 'aeaaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''

```

Optional parameters and arguments

A method definition can specify that its parameters are required or that they are optional. By default, parameters are required. Optional parameters are specified by including the parameter's default value in the method definition. When the method is called, if no argument is supplied for an optional parameter, the default value is used instead.

The parameter's default value must be assigned by one of the following kinds of expressions:

- A constant, such as a literal string or number.
- An expression of the form `new ValType()`, where `ValType` is a value type. Note that this invokes the value type's implicit parameterless constructor, which is not an actual member of the type.
- An expression of the form `default(ValType)`, where `ValType` is a value type.

If a method includes both required and optional parameters, optional parameters are defined at the end of the parameter list, after all required parameters.

The following example defines a method, `ExampleMethod`, that has one required and two optional parameters.

```

using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
    {
        Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                          optionalInt, required + optionalInt);
    }
}

```

If a method with multiple optional arguments is invoked using positional arguments, the caller must supply an argument for all optional parameters from the first one to the last one for which an argument is supplied. In the case of the `ExampleMethod` method, for example, if the caller supplies an argument for the `description` parameter, it must also supply one for the `optionalInt` parameter.

`opt.ExampleMethod(2, 2, "Addition of 2 and 2");` is a valid method call;

`opt.ExampleMethod(2, , "Addition of 2 and 0");` generates an "Argument missing" compiler error.

If a method is called using named arguments or a combination of positional and named arguments, the caller can omit any arguments that follow the last positional argument in the method call.

The following example calls the `ExampleMethod` method three times. The first two method calls use positional arguments. The first omits both optional arguments, while the second omits the last argument. The third method call supplies a positional argument for the required parameter but uses a named argument to supply a value to the `description` parameter while omitting the `optionalInt` argument.

```

public class Example
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}

// The example displays the following output:
//      Optional Description: 10 + 0 = 10
//      Optional Description: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12

```

The use of optional parameters affects *overload resolution*, or the way in which the C# compiler determines which particular overload should be invoked by a method call, as follows:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that does not have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) is not `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a variable, constant, or expression that matches the return type will return that value to the method caller. Methods with a non-void return type are required to use the `return` keyword to return a value. The `return` keyword also stops the execution of the method.

If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block.

For example, these two methods use the `return` keyword to return integers:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

Sometimes, you want your method to return more than a single value. Starting with C# 7.0, you can do this easily by using *tuple types* and *tuple literals*. The tuple type defines the data types of the tuple's elements. Tuple literals provide the actual values of the returned tuple. In the following example, `(string, string, string, int)` defines the tuple type that is returned by the `GetPersonalInfo` method. The expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` is the tuple literal; the method returns the first, middle, and last name, along with the age, of a `PersonInfo` object.

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The caller can then consume the returned tuple with code like the following:

```
var person = GetPersonalInfo("111111111")
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Names can also be assigned to the tuple elements in the tuple type definition. The following example shows an alternate version of the `GetPersonalInfo` method that uses named elements:

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The previous call to the `GetPersonalInfo` method can then be modified as follows:

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

If a method is passed an array as an argument and modifies the value of individual elements, it is not necessary for the method to return the array, although you may choose to do so for good style or functional flow of values. This is because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the following example, changes to the contents of the `values` array that are made in the `DoubleValues` method are observable by any code that has a reference to the array.

```
using System;

public class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8 };
        DoubleValues(values);
        foreach (var value in values)
            Console.Write("{0} ", value);
    }

    public static void DoubleValues(int[] arr)
    {
        for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
            arr[ctr] = arr[ctr] * 2;
    }
}
// The example displays the following output:
//      4 8 12 16
```

Extension methods

Ordinarily, there are two ways to add a method to an existing type:

- Modify the source code for that type. You cannot do this, of course, if you do not own the type's source code. And this becomes a breaking change if you also add any private data fields to support the method.
- Define the new method in a derived class. A method cannot be added in this way using inheritance for other types, such as structures and enumerations. Nor can it be used to "add" a method to a sealed class.

Extension methods let you "add" a method to an existing type without modifying the type itself or implementing the new method in an inherited type. The extension method also does not have to reside in the same assembly

as the type it extends. You call an extension method as if it were a defined member of a type.

For more information, see [Extension Methods](#).

Async Methods

By using the `async` feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the `async` modifier, you can use the `await` operator in the method. When control reaches an `await` expression in the `async` method, control returns to the caller if the awaited task is not completed, and progress in the method with the `await` keyword is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

NOTE

An `async` method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the `async` method, whichever occurs first.

An `async` method typically has a return type of `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An `async` method that returns `void` can't be awaited, and the caller of a `void`-returning method can't catch exceptions that the method throws. Starting with C# 7.0, an `async` method can have [any task-like return type](#).

In the following example, `DelayAsync` is an `async` method that has a return statement that returns an integer. Because it is an `async` method, its method declaration must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer, as the following `int result = await delayTask` statement demonstrates.

```
using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

An `async` method can't declare any `in`, `ref`, or `out` parameters, but it can call methods that have such parameters.

For more information about async methods, see [Asynchronous programming with async and await](#) and [Async return types](#).

Expression-bodied members

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using

=> :

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read-only, and you do not use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the `yield return` statement to return each element one at a time. When a `yield return` statement is reached, the current location is remembered so that the caller can request the next element in the sequence.

The return type of an iterator can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

For more information, see [Iterators](#).

See also

- [Access Modifiers](#)
- [Static Classes and Static Class Members](#)
- [Inheritance](#)
- [Abstract and Sealed Classes and Class Members](#)
- [params](#)
- [out](#)
- [ref](#)
- [in](#)
- [Passing Parameters](#)

Properties

3/6/2021 • 9 minutes to read • [Edit Online](#)

Properties are first class citizens in C#. The language defines syntax that enables developers to write code that accurately expresses their design intent.

Properties behave like fields when they are accessed. However, unlike fields, properties are implemented with accessors that define the statements executed when a property is accessed or assigned.

Property syntax

The syntax for properties is a natural extension to fields. A field defines a storage location:

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

A property definition contains declarations for a `get` and `set` accessor that retrieves and assigns the value of that property:

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

The syntax shown above is the *auto property* syntax. The compiler generates the storage location for the field that backs up the property. The compiler also implements the body of the `get` and `set` accessors.

Sometimes, you need to initialize a property to a value other than the default for its type. C# enables that by setting a value after the closing brace for the property. You may prefer the initial value for the `FirstName` property to be the empty string rather than `null`. You would specify that as shown below:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

Specific initialization is most useful for read-only properties, as you'll see later in this article.

You can also define the storage yourself, as shown below:

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

When a property implementation is a single expression, you can use *expression-bodied members* for the getter or setter:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

This simplified syntax will be used where applicable throughout this article.

The property definition shown above is a read-write property. Notice the keyword `value` in the set accessor. The `set` accessor always has a single parameter named `value`. The `get` accessor must return a value that is convertible to the type of the property (`string` in this example).

That's the basics of the syntax. There are many different variations that support a variety of different design idioms. Let's explore, and learn the syntax options for each.

Scenarios

The examples above showed one of the simplest cases of property definition: a read-write property with no validation. By writing the code you want in the `get` and `set` accessors, you can create many different scenarios.

Validation

You can write code in the `set` accessor to ensure that the values represented by a property are always valid. For example, suppose one rule for the `Person` class is that the name cannot be blank or white space. You would write that as follows:

```

public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}

```

The preceding example can be simplified by using a `throw` expression as part of the property setter validation:

```

public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrWhiteSpace(value)) ? value : throw new ArgumentException("First
name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}

```

The example above enforces the rule that the first name must not be blank or white space. If a developer writes

```
hero.FirstName = "";
```

That assignment throws an `ArgumentException`. Because a property set accessor must have a void return type, you report errors in the set accessor by throwing an exception.

You can extend this same syntax to anything needed in your scenario. You can check the relationships between different properties, or validate against any external conditions. Any valid C# statements are valid in a property accessor.

Read-only

Up to this point, all the property definitions you have seen are read/write properties with public accessors.

That's not the only valid accessibility for properties. You can create read-only properties, or give different accessibility to the set and get accessors. Suppose that your `Person` class should only enable changing the value of the `FirstName` property from other methods in that class. You could give the set accessor `private` accessibility instead of `public`:

```

public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}

```

Now, the `FirstName` property can be accessed from any code, but it can only be assigned from other code in the `Person` class.

You can add any restrictive access modifier to either the set or get accessors. Any access modifier you place on the individual accessor must be more limited than the access modifier on the property definition. The above is legal because the `FirstName` property is `public`, but the set accessor is `private`. You could not declare a `private` property with a `public` accessor. Property declarations can also be declared `protected`, `internal`, `protected internal`, or even `private`.

It is also legal to place the more restrictive modifier on the `get` accessor. For example, you could have a `public` property, but restrict the `get` accessor to `private`. That scenario is rarely done in practice.

You can also restrict modifications to a property so that it can only be set in a constructor or a property initializer. You can modify the `Person` class so as follows:

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

This feature is most commonly used for initializing collections that are exposed as read-only properties:

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
```

Computed properties

A property does not need to simply return the value of a member field. You can create properties that return a computed value. Let's expand the `Person` object to return the full name, computed by concatenating the first and last names:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
```

The example above uses the [string interpolation](#) feature to create the formatted string for the full name.

You can also use an *expression-bodied member*, which provides a more succinct way to create the computed `FullName` property:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Expression-bodied members use the *lambda expression* syntax to define methods that contain a single

expression. Here, that expression returns the full name for the person object.

Cached evaluated properties

You can mix the concept of a computed property with storage and create a *cached evaluated property*. For example, you could update the `FullName` property so that the string formatting only happened the first time it was accessed:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

The above code contains a bug though. If code updates the value of either the `FirstName` or `LastName` property, the previously evaluated `fullName` field is invalid. You modify the `set` accessors of the `FirstName` and `LastName` property so that the `fullName` field is calculated again:

```

public class Person
{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            firstName = value;
            fullName = null;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        set
        {
            lastName = value;
            fullName = null;
        }
    }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}

```

This final version evaluates the `FullName` property only when needed. If the previously calculated version is valid, it's used. If another state change invalidates the previously calculated version, it will be recalculated. Developers that use this class do not need to know the details of the implementation. None of these internal changes affect the use of the Person object. That's the key reason for using Properties to expose data members of an object.

Attaching attributes to auto-implemented properties

Beginning with C# 7.3, field attributes can be attached to the compiler generated backing field in auto-implemented properties. For example, consider a revision to the `Person` class that adds a unique integer `Id` property. You write the `Id` property using an auto-implemented property, but your design does not call for persisting the `Id` property. The `NonSerializedAttribute` can only be attached to fields, not properties. You can attach the `NonSerializedAttribute` to the backing field for the `Id` property by using the `field:` specifier on the attribute, as shown in the following example:

```

public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}

```

This technique works for any attribute you attach to the backing field on the auto-implemented property.

Implementing `INotifyPropertyChanged`

A final scenario where you need to write code in a property accessor is to support the `INotifyPropertyChanged` interface used to notify data binding clients that a value has changed. When the value of a property changes, the object raises the `INotifyPropertyChanged.PropertyChanged` event to indicate the change. The data binding libraries, in turn, update display elements based on that change. The code below shows how you would implement `INotifyPropertyChanged` for the `FirstName` property of this person class.

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
            firstName = value;
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

The `?.` operator is called the *null conditional operator*. It checks for a null reference before evaluating the right side of the operator. The end result is that if there are no subscribers to the `PropertyChanged` event, the code to raise the event doesn't execute. It would throw a `NullReferenceException` without this check in that case. For more information, see [events](#). This example also uses the new `nameof` operator to convert from the property name symbol to its text representation. Using `nameof` can reduce errors where you have mistyped the name of the property.

Again, implementing `INotifyPropertyChanged` is an example of a case where you can write code in your accessors to support the scenarios you need.

Summing up

Properties are a form of smart fields in a class or object. From outside the object, they appear like fields in the object. However, properties can be implemented using the full palette of C# functionality. You can provide validation, different accessibility, lazy evaluation, or any requirements your scenarios need.

Indexers

3/23/2021 • 9 minutes to read • [Edit Online](#)

Indexers are similar to properties. In many ways indexers build on the same language features as [properties](#).

Indexers enable *indexed* properties: properties referenced using one or more arguments. Those arguments provide an index into some collection of values.

Indexer Syntax

You access an indexer through a variable name and square brackets. You place the indexer arguments inside the brackets:

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

You declare indexers using the `this` keyword as the property name, and declaring the arguments within square brackets. This declaration would match the usage shown in the previous paragraph:

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

From this initial example, you can see the relationship between the syntax for properties and for indexers. This analogy carries through most of the syntax rules for indexers. Indexers can have any valid access modifiers (public, protected internal, protected, internal, private or private protected). They may be sealed, virtual, or abstract. As with properties, you can specify different access modifiers for the get and set accessors in an indexer. You may also specify read-only indexers (by omitting the set accessor), or write-only indexers (by omitting the get accessor).

You can apply almost everything you learn from working with properties to indexers. The only exception to that rule is *auto implemented properties*. The compiler cannot always generate the correct storage for an indexer.

The presence of arguments to reference an item in a set of items distinguishes indexers from properties. You may define multiple indexers on a type, as long as the argument lists for each indexer is unique. Let's explore different scenarios where you might use one or more indexers in a class definition.

Scenarios

You would define *indexers* in your type when its API models some collection where you define the arguments to that collection. Your indexers may or may not map directly to the collection types that are part of the .NET core framework. Your type may have other responsibilities in addition to modeling a collection. Indexers enable you to provide the API that matches your type's abstraction without exposing the inner details of how the values for that abstraction are stored or computed.

Let's walk through some of the common scenarios for using *indexers*. You can access the [sample folder for indexers](#). For download instructions, see [Samples and Tutorials](#).

Arrays and Vectors

One of the most common scenarios for creating indexers is when your type models an array, or a vector. You can

create an indexer to model an ordered list of data.

The advantage of creating your own indexer is that you can define the storage for that collection to suit your needs. Imagine a scenario where your type models historical data that is too large to load into memory at once. You need to load and unload sections of the collection based on usage. The example following models this behavior. It reports on how many data points exist. It creates pages to hold sections of the data on demand. It removes pages from memory to make room for pages needed by more recent requests.

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }

        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
        }

        public bool Dirty => dirty;
        public DateTime LastAccess => lastAccess;
    }

    private readonly int totalSize;
    private readonly List<Page> pagesInMemory = new List<Page>();

    public DataSamples(int totalSize)
    {
        this.totalSize = totalSize;
    }
}
```

```

        }

        public Measurements this[int index]
        {
            get
            {
                if (index < 0)
                    throw new IndexOutOfRangeException("Cannot index less than 0");
                if (index >= totalSize)
                    throw new IndexOutOfRangeException("Cannot index past the end of storage");

                var page = updateCachedPagesForAccess(index);
                return page[index];
            }
            set
            {
                if (index < 0)
                    throw new IndexOutOfRangeException("Cannot index less than 0");
                if (index >= totalSize)
                    throw new IndexOutOfRangeException("Cannot index past the end of storage");
                var page = updateCachedPagesForAccess(index);

                page[index] = value;
            }
        }

        private Page updateCachedPagesForAccess(int index)
        {
            foreach (var p in pagesInMemory)
            {
                if (p.HasItem(index))
                {
                    return p;
                }
            }
            var startingIndex = (index / 1000) * 1000;
            var newPassword = new Page(startingIndex, 1000);
            addPageToCache(newPassword);
            return newPassword;
        }

        private void addPageToCache(Page p)
        {
            if (pagesInMemory.Count > 4)
            {
                // remove oldest non-dirty page:
                var oldest = pagesInMemory
                    .Where(page => !page.Dirty)
                    .OrderBy(page => page.LastAccess)
                    .FirstOrDefault();
                // Note that this may keep more than 5 pages in memory
                // if too much is dirty
                if (oldest != null)
                    pagesInMemory.Remove(oldest);
            }
            pagesInMemory.Add(p);
        }
    }
}

```

You can follow this design idiom to model any sort of collection where there are good reasons not to load the entire set of data into an in- memory collection. Notice that the `Page` class is a private nested class that is not part of the public interface. Those details are hidden from any users of this class.

Dictionaries

Another common scenario is when you need to model a dictionary or a map. This scenario is when your type stores values based on key, typically text keys. This example creates a dictionary that maps command line

arguments to [lambda expressions](#) that manage those options. The following example shows two classes: an `ArgsActions` class that maps a command line option to an `Action` delegate, and an `ArgsProcessor` that uses the `ArgsActions` to execute each `Action` when it encounters that option.

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }
}

public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

In this example, the `ArgsAction` collection maps closely to the underlying collection. The `get` determines if a given option has been configured. If so, it returns the `Action` associated with that option. If not, it returns an `Action` that does nothing. The public accessor does not include a `set` accessor. Rather, the design using a public method for setting options.

Multi-Dimensional Maps

You can create indexers that use multiple arguments. In addition, those arguments are not constrained to be the same type. Let's look at two examples.

The first example shows a class that generates values for a Mandelbrot set. For more information on the mathematics behind the set, read [this article](#). The indexer uses two doubles to define a point in the X, Y plane. The get accessor computes the number of iterations until a point is determined to be not in the set. If the maximum iterations is reached, the point is in the set, and the class's `maxIterations` value is returned. (The computer generated images popularized for the Mandelbrot set define colors for the number of iterations necessary to determine that a point is outside the set.

```

public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                   (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}

```

The Mandelbrot Set defines values at every (x,y) coordinate for real number values. That defines a dictionary that could contain an infinite number of values. Therefore, there is no storage behind the set. Instead, this class computes the value for each point when code calls the `get` accessor. There's no underlying storage used.

Let's examine one last use of indexers, where the indexer takes multiple arguments of different types. Consider a program that manages historical temperature data. This indexer uses a city and a date to set or get the high and low temperatures for that location:

```

using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // Strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}

```

This example creates an indexer that maps weather data on two different arguments: a city (represented by a `string`) and a date (represented by a `DateTime`). The internal storage uses two `Dictionary` classes to represent the two-dimensional dictionary. The public API no longer represents the underlying storage. Rather, the language features of indexers enables you to create a public interface that represents your abstraction, even though the underlying storage must use different core collection types.

There are two parts of this code that may be unfamiliar to some developers. These two `using` directives:

```

using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;

```

create an *alias* for a constructed generic type. Those statements enable the code later to use the more descriptive `DateMeasurements` and `CityDataMeasurements` names instead of the generic construction of `Dictionary<DateTime, Measurements>` and `Dictionary<string, Dictionary<DateTime, Measurements>>`. This construct does require using the fully qualified type names on the right side of the `=` sign.

The second technique is to strip off the time portions of any `DateTime` object used to index into the collections.

.NET doesn't include a date-only type. Developers use the `DateTime` type, but use the `Date` property to ensure that any `DateTime` object from that day are equal.

Summing Up

You should create indexers anytime you have a property-like element in your class where that property represents not a single value, but rather a collection of values where each individual item is identified by a set of arguments. Those arguments can uniquely identify which item in the collection should be referenced. Indexers extend the concept of [properties](#), where a member is treated like a data item from outside the class, but like a method on the inside. Indexers allow arguments to find a single item in a property that represents a set of items.

Discards - C# Guide

3/6/2021 • 8 minutes to read • [Edit Online](#)

Starting with C# 7.0, C# supports discards, which are placeholder variables that are intentionally unused in application code. Discards are equivalent to unassigned variables; they don't have a value. A discard communicates intent to the compiler and others that read your code: You intended to ignore the result of an expression. You may want to ignore the result of an expression, one or more members of a tuple expression, an `out` parameter to a method, or the target of a pattern matching expression.

Because there's only a single discard variable, that variable may not even be allocated storage. Discards can reduce memory allocations. Discards make the intent of your code clear. They enhance its readability and maintainability.

You indicate that a variable is a discard by assigning it the underscore (`_`) as its name. For example, the following method call returns a tuple in which the first and second values are discards. `area` is a previously declared variable set to the third component returned by `GetCityInformation`:

```
(_, _, area) = city.GetCityInformation(cityName);
```

Beginning with C# 9.0, you can use discards to specify unused input parameters of a lambda expression. For more information, see the [Input parameters of a lambda expression](#) section of the [Lambda expressions](#) article.

When `_` is a valid discard, attempting to retrieve its value or use it in an assignment operation generates compiler error CS0301, "The name '_' doesn't exist in the current context". This error is because `_` isn't assigned a value, and may not even be assigned a storage location. If it were an actual variable, you couldn't discard more than one value, as the previous example did.

Tuple and object deconstruction

Discards are useful in working with tuples when your application code uses some tuple elements but ignores others. For example, the following `queryCityDataForYears` method returns a tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

```
var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

For more information on deconstructing tuples with discards, see [Deconstructing tuples and other types](#).

The `Deconstruct` method of a class, structure, or interface also allows you to retrieve and deconstruct a specific set of data from an object. You can use discards when you're interested in working with only a subset of the deconstructed values. The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state), but discards the last name and the state.

```

using System;

namespace Discards
{
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public Person(string fname, string mname, string lname,
                      string cityName, string stateName)
        {
            FirstName = fname;
            MiddleName = mname;
            LastName = lname;
            City = cityName;
            State = stateName;
        }

        // Return the first and last name.
        public void Deconstruct(out string fname, out string lname)
        {
            fname = FirstName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string mname, out string lname)
        {
            fname = FirstName;
            mname = MiddleName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string lname,
                      out string city, out string state)
        {
            fname = FirstName;
            lname = LastName;
            city = City;
            state = State;
        }
    }

    class Example
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fName, _, city, _) = p;
            Console.WriteLine($"Hello {fName} of {city}!");
            // The example displays the following output:
            //      Hello John of Boston!
        }
    }
}

```

For more information on deconstructing user-defined types with discards, see [Deconstructing tuples and other types](#).

Pattern matching with `switch`

The *discard pattern* can be used in pattern matching with the [switch expression](#). Every expression, including `null`, always matches the discard pattern.

The following example defines a `ProvidesFormatInfo` method that uses a `switch` expression to determine whether an object provides an [IFormatProvider](#) implementation and tests whether the object is `null`. It also uses the discard pattern to handle non-null objects of any other type.

```
object[] objects = { CultureInfo.CurrentCulture,
                     CultureInfo.CurrentCulture.DateTimeFormat,
                     CultureInfo.CurrentCulture.NumberFormat,
                     new ArgumentException(), null };
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
//   System.Globalization.CultureInfo object
//   System.Globalization.DateTimeFormatInfo object
//   System.Globalization.NumberFormatInfo object
//   Some object type without format information
//   A null object reference: Its use could result in a NullReferenceException
```

Calls to methods with `out` parameters

When calling the `Deconstruct` method to deconstruct a user-defined type (an instance of a class, structure, or interface), you can discard the values of individual `out` arguments. But you can also discard the value of `out` arguments when calling any method with an `out` parameter.

The following example calls the [DateTime.TryParse\(String, out DateTime\)](#) method to determine whether the string representation of a date is valid in the current culture. Because the example is concerned only with validating the date string and not with parsing it to extract the date, the `out` argument to the method is a discard.

```
string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                       "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                       "5/01/2018 14:57:32.80 -07:00",
                       "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                       "Fri, 15 May 2018 20:10:57 GMT" };
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
//   '05/01/2018 14:57:32.8': valid
//   '2018-05-01 14:57:32.8': valid
//   '2018-05-01T14:57:32.8375298-04:00': valid
//   '5/01/2018': valid
//   '5/01/2018 14:57:32.80 -07:00': valid
//   '1 May 2018 2:57:32.8 PM': valid
//   '16-05-2018 1:00:32 PM': invalid
//   'Fri, 15 May 2018 20:10:57 GMT': invalid
```

A standalone discard

You can use a standalone discard to indicate any variable that you choose to ignore. One typical use is to use an assignment to ensure that an argument isn't null. The following code uses a discard to force an assignment. The right side of the assignment uses the [null coalescing operator](#) to throw an [System.ArgumentNullException](#) when the argument is `null`. The code doesn't need the result of the assignment, so it's discarded. The expression forces a null check. The discard clarifies your intent: the result of the assignment isn't needed or used.

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg), message: "arg can't be null");

    // Do work with arg.
}
```

The following example uses a standalone discard to ignore the `Task` object returned by an asynchronous operation. Assigning the task has the effect of suppressing the exception that the operation throws as it is about to complete. It makes your intent clear: You want to discard the `Task`, and ignore any errors generated from that asynchronous operation.

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//      About to launch a task...
//      Completed looping operation...
//      Exiting after 5 second delay
```

Without assigning the task to a discard, the following code generates a compiler warning:

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current method continues before the call
    // is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
```

NOTE

If you run either of the preceding two samples using a debugger, the debugger will stop the program when the exception is thrown. Without a debugger attached, the exception is silently ignored in both cases.

`_` is also a valid identifier. When used outside of a supported context, `_` is treated not as a discard but as a valid variable. If an identifier named `_` is already in scope, the use of `_` as a standalone discard can result in:

- Accidental modification of the value of the in-scope `_` variable by assigning it the value of the intended discard. For example:

```
private static void ShowValue(int _)
{
    byte[] arr = { 0, 0, 1, 2 };
    _ = BitConverter.ToInt32(arr, 0);
    Console.WriteLine(_);
}
// The example displays the following output:
//      33619968
```

- A compiler error for violating type safety. For example:

```
private static bool RoundTrips(int _)
{
    string value = _.ToString();
    int newValue = 0;
    _ = Int32.TryParse(value, out newValue);
    return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- Compiler error CS0136, "A local or parameter named '`_`' cannot be declared in this scope because that name is used in an enclosing local scope to define a local or parameter." For example:

```
public void DoSomething(int _)
{
    var _ = GetValue(); // Error: cannot declare local _ when one is already in scope
}
// The example displays the following compiler error:
// error CS0136:
//      A local or parameter named '_' cannot be declared in this scope
//      because that name is used in an enclosing local scope
//      to define a local or parameter
```

See also

- [Deconstructing tuples and other types](#)
- [is keyword](#)
- [switch keyword](#)

Generics (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Generics introduce the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter `T`, you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }

    class TestGenericList
    {
        private class ExampleClass { }

        static void Main()
        {
            // Declare a list of type int.
            GenericList<int> list1 = new GenericList<int>();
            list1.Add(1);

            // Declare a list of type string.
            GenericList<string> list2 = new GenericList<string>();
            list2.Add("");

            // Declare a list of type ExampleClass.
            GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
            list3.Add(new ExampleClass());
        }
    }
}
```

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. The [System.Collections.Generic](#) namespace contains several generic-based collection classes. The non-generic collections, such as [ArrayList](#) are not recommended and are maintained for compatibility purposes. For more information, see [Generics in .NET](#).

Of course, you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the `List<T>` class provided by .NET instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

Note that `T` is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

```

// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumarator<T> GetEnumarator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}

```

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Generics overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET class library contains several generic collection classes in the [System.Collections.Generic](#) namespace. These should be used whenever possible instead of classes such as [ArrayList](#) in the [System.Collections](#) namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related sections

- [Generic Type Parameters](#)
- [Constraints on Type Parameters](#)
- [Generic Classes](#)
- [Generic Interfaces](#)
- [Generic Methods](#)
- [Generic Delegates](#)
- [Differences Between C++ Templates and C# Generics](#)
- [Generics and Reflection](#)
- [Generics in the Run Time](#)

C# language specification

For more information, see the [C# Language Specification](#).

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Types](#)
- [<typeparam>](#)

- <typeparamref>
- Generics in .NET

Iterators

3/23/2021 • 5 minutes to read • [Edit Online](#)

Almost every program you write will have some need to iterate over a collection. You'll write code that examines every item in a collection.

You'll also create iterator methods which are methods that produces an iterator (which is an object that traverses a container, particularly lists) for the elements of that class. These can be used for:

- Performing an action on each item in a collection.
- Enumerating a custom collection.
- Extending [LINQ](#) or other libraries.
- Creating a data pipeline where data flows efficiently through iterator methods.

The C# language provides features for both these scenarios. This article provides an overview of those features.

This tutorial has multiple steps. After each step, you can run the application and see the progress. You can also [view or download the completed sample](#) for this topic. For download instructions, see [Samples and Tutorials](#).

Iterating with foreach

Enumerating a collection is simple: The `foreach` keyword enumerates a collection, executing the embedded statement once for each element in the collection:

```
foreach (var item in collection)
{
    Console.WriteLine(item.ToString());
}
```

That's all there is to it. To iterate over all the contents of a collection, the `foreach` statement is all you need. The `foreach` statement isn't magic, though. It relies on two generic interfaces defined in the .NET core library in order to generate the code necessary to iterate a collection: `IEnumerable<T>` and `IEnumerator<T>`. This mechanism is explained in more detail below.

Both of these interfaces also have non-generic counterparts: `IEnumerable` and `IEnumerator`. The [generic](#) versions are preferred for modern code.

Enumeration sources with iterator methods

Another great feature of the C# language enables you to build methods that create a source for an enumeration. These are referred to as *iterator methods*. An iterator method defines how to generate the objects in a sequence when requested. You use the `yield return` contextual keywords to define an iterator method.

You could write this method to produce the sequence of integers from 0 through 9:

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}

```

The code above shows distinct `yield return` statements to highlight the fact that you can use multiple discrete `yield return` statements in an iterator method. You can (and often do) use other language constructs to simplify the code of an iterator method. The method definition below produces the exact same sequence of numbers:

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}

```

You don't have to decide one or the other. You can have as many `yield return` statements as necessary to meet the needs of your method:

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}

```

That's the basic syntax. Let's consider a real world example where you would write an iterator method. Imagine you're on an IoT project and the device sensors generate a very large stream of data. To get a feel for the data, you might write a method that samples every Nth data element. This small iterator method does the trick:

```

public static IEnumerable<T> Sample(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}

```

There is one important restriction on iterator methods: you can't have both a `return` statement and a `yield return` statement in the same method. The following will not compile:

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}

```

This restriction normally isn't a problem. You have a choice of either using `yield return` throughout the method, or separating the original method into multiple methods, some using `return`, and some using `yield return`.

You can modify the last method slightly to use `yield return` everywhere:

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}

```

Sometimes, the right answer is to split an iterator method into two different methods. One that uses `return`, and a second that uses `yield return`. Consider a situation where you might want to return an empty collection, or the first 5 odd numbers, based on a boolean argument. You could write that as these two methods:

```

public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}

```

Look at the methods above. The first uses the standard `return` statement to return either an empty collection, or the iterator created by the second method. The second method uses the `yield return` statement to create the requested sequence.

Deeper Dive into `foreach`

The `foreach` statement expands into a standard idiom that uses the `IEnumerable<T>` and `IEnumerator<T>` interfaces to iterate across all elements of a collection. It also minimizes errors developers make by not properly managing resources.

The compiler translates the `foreach` loop shown in the first example into something similar to this construct:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

The construct above represents the code generated by the C# compiler as of version 5 and above. Prior to version 5, the `item` variable had a different scope:

```
// C# versions 1 through 4:
IEnumerator<int> enumerator = collection.GetEnumerator();
int item = default(int);
while (enumerator.MoveNext())
{
    item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

This was changed because the earlier behavior could lead to subtle and hard to diagnose bugs involving lambda expressions. For more information about lambda expressions, see [Lambda expressions](#).

The exact code generated by the compiler is somewhat more complicated, and handles situations where the object returned by `GetEnumerator()` implements the `IDisposable` interface. The full expansion generates code more like this:

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    } finally
    {
        // dispose of enumerator.
    }
}
```

The manner in which the enumerator is disposed of depends on the characteristics of the type of `enumerator`. In the general case, the `finally` clause expands to:

```
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

However, if the type of `enumerator` is a sealed type and there is no implicit conversion from the type of `enumerator` to `IDisposable`, the `finally` clause expands to an empty block:

```
finally
{
}
```

If there is an implicit conversion from the type of `enumerator` to `IDisposable`, and `enumerator` is a non-nullable value type, the `finally` clause expands to:

```
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

Thankfully, you don't need to remember all these details. The `foreach` statement handles all those nuances for you. The compiler will generate the correct code for any of these constructs.

Introduction to Delegates

3/6/2021 • 2 minutes to read • [Edit Online](#)

Delegates provide a *late binding* mechanism in .NET. Late Binding means that you create an algorithm where the caller also supplies at least one method that implements part of the algorithm.

For example, consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness.

In all those cases, the Sort() method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings.

These kinds of solutions have been used in software for half a century. The C# language delegate concept provides first class language support, and type safety around the concept.

As you'll see later in this series, the C# code you write for algorithms like this is type safe, and uses the language rules and the compiler to ensure that the types match for arguments and return types.

[Function pointers](#) were added to C# 9 for similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked using a virtual method added to a delegate type. Using function pointers, you can specify different conventions.

Language Design Goals for Delegates

The language designers enumerated several goals for the feature that eventually became delegates.

The team wanted a common language construct that could be used for any late binding algorithms. Delegates enable developers to learn one concept, and use that same concept across many different software problems.

Second, the team wanted to support both single and multicast method calls. (Multicast delegates are delegates that chain together multiple method calls. You'll see examples [later in this series](#).)

The team wanted delegates to support the same type safety that developers expect from all C# constructs.

Finally, the team recognized an event pattern is one specific pattern where delegates, or any late binding algorithm, is very useful. The team wanted to ensure the code for delegates could provide the basis for the .NET event pattern.

The result of all that work was the delegate and event support in C# and .NET. The remaining articles in this section will cover language features, library support, and common idioms used when you work with delegates.

You'll learn about the `delegate` keyword and what code it generates. You'll learn about the features in the `System.Delegate` class, and how those features are used. You'll learn how to create type safe delegates, and how to create methods that can be invoked through delegates. You'll also learn how to work with delegates and events by using Lambda expressions. You'll see where delegates become one of the building blocks for LINQ. You'll learn how delegates are the basis for the .NET event pattern, and how they're different.

Let's get started.

[Next](#)

System.Delegate and the `delegate` keyword

11/2/2020 • 6 minutes to read • [Edit Online](#)

[Previous](#)

This article covers the classes in .NET that support delegates, and how those map to the `delegate` keyword.

Define delegate types

Let's start with the 'delegate' keyword, because that's primarily what you will use as you work with delegates. The code that the compiler generates when you use the `delegate` keyword will map to method calls that invoke members of the [Delegate](#) and [MulticastDelegate](#) classes.

You define a delegate type using syntax that is similar to defining a method signature. You just add the `delegate` keyword to the definition.

Let's continue to use the `List.Sort()` method as our example. The first step is to create a type for the comparison delegate:

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

The compiler generates a class, derived from [System.Delegate](#) that matches the signature used (in this case, a method that returns an integer, and has two arguments). The type of that delegate is [Comparison](#). The [Comparison](#) delegate type is a generic type. For details on generics see [here](#).

Notice that the syntax may appear as though it is declaring a variable, but it is actually declaring a *type*. You can define delegate types inside classes, directly inside namespaces, or even in the global namespace.

NOTE

Declaring delegate types (or other types) directly in the global namespace is not recommended.

The compiler also generates add and remove handlers for this new type so that clients of this class can add and remove methods from an instance's invocation list. The compiler will enforce that the signature of the method being added or removed matches the signature used when declaring the method.

Declare instances of delegates

After defining the delegate, you can create an instance of that type. Like all variables in C#, you cannot declare delegate instances directly in a namespace, or in the global namespace.

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

The type of the variable is [Comparison<T>](#), the delegate type defined earlier. The name of the variable is

```
comparator .
```

That code snippet above declared a member variable inside a class. You can also declare delegate variables that are local variables, or arguments to methods.

Invoke delegates

You invoke the methods that are in the invocation list of a delegate by calling that delegate. Inside the `Sort()` method, the code will call the comparison method to determine which order to place objects:

```
int result = comparator(left, right);
```

In the line above, the code *invokes* the method attached to the delegate. You treat the variable as a method name, and invoke it using normal method call syntax.

That line of code makes an unsafe assumption: There's no guarantee that a target has been added to the delegate. If no targets have been attached, the line above would cause a `NullReferenceException` to be thrown. The idioms used to address this problem are more complicated than a simple null-check, and are covered later in this [series](#).

Assign, add, and remove invocation targets

That's how a delegate type is defined, and how delegate instances are declared and invoked.

Developers that want to use the `List.Sort()` method need to define a method whose signature matches the delegate type definition, and assign it to the delegate used by the sort method. This assignment adds the method to the invocation list of that delegate object.

Suppose you wanted to sort a list of strings by their length. Your comparison function might be the following:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

The method is declared as a private method. That's fine. You may not want this method to be part of your public interface. It can still be used as the comparison method when attached to a delegate. The calling code will have this method attached to the target list of the delegate object, and can access it through that delegate.

You create that relationship by passing that method to the `List.Sort()` method:

```
phrases.Sort(CompareLength);
```

Notice that the method name is used, without parentheses. Using the method as an argument tells the compiler to convert the method reference into a reference that can be used as a delegate invocation target, and attach that method as an invocation target.

You could also have been explicit by declaring a variable of type `Comparison<string>` and doing an assignment:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

In uses where the method being used as a delegate target is a small method, it's common to use [lambda expression](#) syntax to perform the assignment:

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Using lambda expressions for delegate targets is covered more in a [later section](#).

The Sort() example typically attaches a single target method to the delegate. However, delegate objects do support invocation lists that have multiple target methods attached to a delegate object.

Delegate and MulticastDelegate classes

The language support described above provides the features and support you'll typically need to work with delegates. These features are built on two classes in the .NET Core framework: [Delegate](#) and [MulticastDelegate](#).

The `System.Delegate` class and its single direct subclass, `System.MulticastDelegate`, provide the framework support for creating delegates, registering methods as delegate targets, and invoking all methods that are registered as a delegate target.

Interestingly, the `System.Delegate` and `System.MulticastDelegate` classes are not themselves delegate types. They do provide the basis for all specific delegate types. That same language design process mandated that you cannot declare a class that derives from `Delegate` or `MulticastDelegate`. The C# language rules prohibit it.

Instead, the C# compiler creates instances of a class derived from `MulticastDelegate` when you use the C# language keyword to declare delegate types.

This design has its roots in the first release of C# and .NET. One goal for the design team was to ensure that the language enforced type safety when using delegates. That meant ensuring that delegates were invoked with the right type and number of arguments. And, that any return type was correctly indicated at compile time. Delegates were part of the 1.0 .NET release, which was before generics.

The best way to enforce this type safety was for the compiler to create the concrete delegate classes that represented the method signature being used.

Even though you cannot create derived classes directly, you will use the methods defined on these classes. Let's go through the most common methods that you will use when you work with delegates.

The first, most important fact to remember is that every delegate you work with is derived from `MulticastDelegate`. A multicast delegate means that more than one method target can be invoked when invoking through a delegate. The original design considered making a distinction between delegates where only one target method could be attached and invoked, and delegates where multiple target methods could be attached and invoked. That distinction proved to be less useful in practice than originally thought. The two different classes were already created, and have been in the framework since its initial public release.

The methods that you will use the most with delegates are `Invoke()` and `BeginInvoke()` / `EndInvoke()`. `Invoke()` will invoke all the methods that have been attached to a particular delegate instance. As you saw above, you typically invoke delegates using the method call syntax on the delegate variable. As you'll see [later in this series](#), there are patterns that work directly with these methods.

Now that you've seen the language syntax and the classes that support delegates, let's examine how strongly typed delegates are used, created, and invoked.

[Next](#)

Strongly Typed Delegates

3/12/2020 • 2 minutes to read • [Edit Online](#)

[Previous](#)

In the previous article, you saw that you create specific delegate types using the `delegate` keyword.

The abstract Delegate class provide the infrastructure for loose coupling and invocation. Concrete Delegate types become much more useful by embracing and enforcing type safety for the methods that are added to the invocation list for a delegate object. When you use the `delegate` keyword and define a concrete delegate type, the compiler generates those methods.

In practice, this would lead to creating new delegate types whenever you need a different method signature. This work could get tedious after a time. Every new feature requires new delegate types.

Thankfully, this isn't necessary. The .NET Core framework contains several types that you can reuse whenever you need delegate types. These are `generic` definitions so you can declare customizations when you need new method declarations.

The first of these types is the `Action` type, and several variations:

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

The `in` modifier on the generic type argument is covered in the article on covariance.

There are variations of the `Action` delegate that contain up to 16 arguments such as `Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>`. It's important that these definitions use different generic arguments for each of the delegate arguments: That gives you maximum flexibility. The method arguments need not be, but may be, the same type.

Use one of the `Action` types for any delegate type that has a void return type.

The framework also includes several generic delegate types that you can use for delegate types that return values:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

The `out` modifier on the result generic type argument is covered in the article on covariance.

There are variations of the `Func` delegate with up to 16 input arguments such as `Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>`. The type of the result is always the last type parameter in all the `Func` declarations, by convention.

Use one of the `Func` types for any delegate type that returns a value.

There's also a specialized `Predicate<T>` type for a delegate that returns a test on a single value:

```
public delegate bool Predicate<in T>(T obj);
```

You may notice that for any `Predicate` type, a structurally equivalent `Func` type exists. For example:

```
Func<string, bool> TestForString;  
Predicate<string> AnotherTestForString;
```

You might think these two types are equivalent. They are not. These two variables cannot be used interchangeably. A variable of one type cannot be assigned the other type. The C# type system uses the names of the defined types, not the structure.

All these delegate type definitions in the .NET Core Library should mean that you do not need to define a new delegate type for any new feature you create that requires delegates. These generic definitions should provide all the delegate types you need under most situations. You can simply instantiate one of these types with the required type parameters. In the case of algorithms that can be made generic, these delegates can be used as generic types.

This should save time, and minimize the number of new types that you need to create in order to work with delegates.

In the next article, you'll see several common patterns for working with delegates in practice.

[Next](#)

Common Patterns for Delegates

3/12/2020 • 8 minutes to read • [Edit Online](#)

[Previous](#)

Delegates provide a mechanism that enables software designs involving minimal coupling between components.

One excellent example for this kind of design is LINQ. The LINQ Query Expression Pattern relies on delegates for all of its features. Consider this simple example:

```
var smallNumbers = numbers.Where(n => n < 10);
```

This filters the sequence of numbers to only those less than the value 10. The `Where` method uses a delegate that determines which elements of a sequence pass the filter. When you create a LINQ query, you supply the implementation of the delegate for this specific purpose.

The prototype for the `Where` method is:

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

This example is repeated with all the methods that are part of LINQ. They all rely on delegates for the code that manages the specific query. This API design pattern is a very powerful one to learn and understand.

This simple example illustrates how delegates require very little coupling between components. You don't need to create a class that derives from a particular base class. You don't need to implement a specific interface. The only requirement is to provide the implementation of one method that is fundamental to the task at hand.

Building Your Own Components with Delegates

Let's build on that example by creating a component using a design that relies on delegates.

Let's define a component that could be used for log messages in a large system. The library components could be used in many different environments, on multiple different platforms. There are a lot of common features in the component that manages the logs. It will need to accept messages from any component in the system. Those messages will have different priorities, which the core component can manage. The messages should have timestamps in their final archived form. For more advanced scenarios, you could filter messages by the source component.

There is one aspect of the feature that will change often: where messages are written. In some environments, they may be written to the error console. In others, a file. Other possibilities include database storage, OS event logs, or other document storage.

There are also combinations of output that might be used in different scenarios. You may want to write messages to the console and to a file.

A design based on delegates will provide a great deal of flexibility, and make it easy to support storage mechanisms that may be added in the future.

Under this design, the primary log component can be a non-virtual, even sealed class. You can plug in any set of delegates to write the messages to different storage media. The built in support for multicast delegates makes it

easy to support scenarios where messages must be written to multiple locations (a file, and a console).

A First Implementation

Let's start small: the initial implementation will accept new messages, and write them using any attached delegate. You can start with one delegate that writes messages to the console.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

The static class above is the simplest thing that can work. We need to write the single implementation for the method that writes messages to the console:

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Finally, you need to hook up the delegate by attaching it to the `WriteMessage` delegate declared in the logger:

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Practices

Our sample so far is fairly simple, but it still demonstrates some of the important guidelines for designs involving delegates.

Using the delegate types defined in the Core Framework makes it easier for users to work with the delegates. You don't need to define new types, and developers using your library do not need to learn new, specialized delegate types.

The interfaces used are as minimal and as flexible as possible: To create a new output logger, you must create one method. That method may be a static method, or an instance method. It may have any access.

Formatting Output

Let's make this first version a bit more robust, and then start creating other logging mechanisms.

Next, let's add a few arguments to the `LogMessage()` method so that your log class creates more structured messages:

```

public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}

```

```

public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}

```

Next, let's make use of that `Severity` argument to filter the messages that are sent to the log's output.

```

public static class Logger
{
    public static Action<string> WriteMessage;

    public static Severity LogLevel {get;set;} = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}

```

Practices

You've added new features to the logging infrastructure. Because the logger component is very loosely coupled to any output mechanism, these new features can be added with no impact on any of the code implementing the logger delegate.

As you keep building this, you'll see more examples of how this loose coupling enables greater flexibility in updating parts of the site without any changes to other locations. In fact, in a larger application, the logger output classes might be in a different assembly, and not even need to be rebuilt.

Building a Second Output Engine

The Log component is coming along well. Let's add one more output engine that logs messages to a file. This will be a slightly more involved output engine. It will be a class that encapsulates the file operations, and ensures that the file is always closed after each write. That ensures that all the data is flushed to disk after each message is generated.

Here is that file based logger:

```

public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}

```

Once you've created this class, you can instantiate it and it attaches its LogMessage method to the Logger component:

```
var file = new FileLogger("log.txt");
```

These two are not mutually exclusive. You could attach both log methods and generate messages to the console and a file:

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the static class we utilized earlier
```

Later, even in the same application, you can remove one of the delegates without any other issues to the system:

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Practices

Now, you've added a second output handler for the logging sub-system. This one needs a bit more infrastructure to correctly support the file system. The delegate is an instance method. It's also a private method. There's no need for greater accessibility because the delegate infrastructure can connect the delegates.

Second, the delegate-based design enables multiple output methods without any extra code. You don't need to build any additional infrastructure to support multiple output methods. They simply become another method on the invocation list.

Pay special attention to the code in the file logging output method. It is coded to ensure that it does not throw any exceptions. While this isn't always strictly necessary, it's often a good practice. If either of the delegate methods throws an exception, the remaining delegates that are on the invocation won't be invoked.

As a last note, the file logger must manage its resources by opening and closing the file on each log message. You could choose to keep the file open and implement `IDisposable` to close the file when you are completed. Either method has its advantages and disadvantages. Both do create a bit more coupling between the classes.

None of the code in the `Logger` class would need to be updated in order to support either scenario.

Handling Null Delegates

Finally, let's update the `LogMessage` method so that it is robust for those cases when no output mechanism is selected. The current implementation will throw a `NullReferenceException` when the `WriteMessage` delegate does not have an invocation list attached. You may prefer a design that silently continues when no methods have been attached. This is easy using the null conditional operator, combined with the `Delegate.Invoke()` method:

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

The null conditional operator (`?.`) short-circuits when the left operand (`WriteMessage` in this case) is null, which means no attempt is made to log a message.

You won't find the `Invoke()` method listed in the documentation for `System.Delegate` or `System.MulticastDelegate`. The compiler generates a type safe `Invoke` method for any delegate type declared. In this example, that means `Invoke` takes a single `string` argument, and has a void return type.

Summary of Practices

You've seen the beginnings of a log component that could be expanded with other writers, and other features. By using delegates in the design these different components are very loosely coupled. This provides several advantages. It's very easy to create new output mechanisms and attach them to the system. These other mechanisms only need one method: the method that writes the log message. It's a design that is very resilient when new features are added. The contract required for any writer is to implement one method. That method could be a static or instance method. It could be public, private, or any other legal access.

The `Logger` class can make any number of enhancements or changes without introducing breaking changes. Like any class, you cannot modify the public API without the risk of breaking changes. But, because the coupling between the logger and any output engines is only through the delegate, no other types (like interfaces or base classes) are involved. The coupling is as small as possible.

[Next](#)

Introduction to events

11/2/2020 • 3 minutes to read • [Edit Online](#)

[Previous](#)

Events are, like delegates, a *late binding* mechanism. In fact, events are built on the language support for delegates.

Events are a way for an object to broadcast (to all interested components in the system) that something has happened. Any other component can subscribe to the event, and be notified when an event is raised.

You've probably used events in some of your programming. Many graphical systems have an event model to report user interaction. These events would report mouse movement, button presses and similar interactions. That's one of the most common, but certainly not the only scenario where events are used.

You can define events that should be raised for your classes. One important consideration when working with events is that there may not be any object registered for a particular event. You must write your code so that it does not raise events when no listeners are configured.

Subscribing to an event also creates a coupling between two objects (the event source, and the event sink). You need to ensure that the event sink unsubscribes from the event source when no longer interested in events.

Design goals for event support

The language design for events targets these goals:

- Enable very minimal coupling between an event source and an event sink. These two components may not be written by the same organization, and may even be updated on totally different schedules.
- It should be very simple to subscribe to an event, and to unsubscribe from that same event.
- Event sources should support multiple event subscribers. It should also support having no event subscribers attached.

You can see that the goals for events are very similar to the goals for delegates. That's why the event language support is built on the delegate language support.

Language support for events

The syntax for defining events, and subscribing or unsubscribing from events is an extension of the syntax for delegates.

To define an event you use the `event` keyword:

```
public event EventHandler<FileListArgs> Progress;
```

The type of the event (`EventHandler<FileListArgs>` in this example) must be a delegate type. There are a number of conventions that you should follow when declaring an event. Typically, the event delegate type has a void return. Event declarations should be a verb, or a verb phrase. Use past tense when the event reports something that has happened. Use a present tense verb (for example, `Closing`) to report something that is about to happen. Often, using present tense indicates that your class supports some kind of customization behavior. One of the most common scenarios is to support cancellation. For example, a `Closing` event may include an argument that would indicate if the close operation should continue, or not. Other scenarios may enable callers

to modify behavior by updating properties of the event arguments. You may raise an event to indicate a proposed next action an algorithm will take. The event handler may mandate a different action by modifying properties of the event argument.

When you want to raise the event, you call the event handlers using the delegate invocation syntax:

```
Progress?.Invoke(this, new FileListArgs(file));
```

As discussed in the section on [Delegates](#), the `?.` operator makes it easy to ensure that you do not attempt to raise the event when there are no subscribers to that event.

You subscribe to an event by using the `+ =` operator:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

The handler method typically has the prefix 'On' followed by the event name, as shown above.

You unsubscribe using the `- =` operator:

```
fileLister.Progress -= onProgress;
```

It's important that you declare a local variable for the expression that represents the event handler. That ensures the unsubscribe removes the handler. If, instead, you used the body of the lambda expression, you are attempting to remove a handler that has never been attached, which does nothing.

In the next article, you'll learn more about typical event patterns, and different variations on this example.

[Next](#)

Standard .NET event patterns

3/12/2020 • 8 minutes to read • [Edit Online](#)

[Previous](#)

.NET events generally follow a few known patterns. Standardizing on these patterns means that developers can leverage knowledge of those standard patterns, which can be applied to any .NET event program.

Let's go through these standard patterns so you will have all the knowledge you need to create standard event sources, and subscribe and process standard events in your code.

Event Delegate Signatures

The standard signature for a .NET event delegate is:

```
void OnEventRaised(object sender, EventArgs args);
```

The return type is void. Events are based on delegates and are multicast delegates. That supports multiple subscribers for any event source. The single return value from a method doesn't scale to multiple event subscribers. Which return value does the event source see after raising an event? Later in this article you'll see how to create event protocols that support event subscribers that report information to the event source.

The argument list contains two arguments: the sender, and the event arguments. The compile time type of `sender` is `System.Object`, even though you likely know a more derived type that would always be correct. By convention, use `object`.

The second argument has typically been a type that is derived from `System.EventArgs`. (You'll see in the [next section](#) that this convention is no longer enforced.) If your event type does not need any additional arguments, you will still provide both arguments. There is a special value, `EventArgs.Empty` that you should use to denote that your event does not contain any additional information.

Let's build a class that lists files in a directory, or any of its subdirectories that follow a pattern. This component raises an event for each file found that matches the pattern.

Using an event model provides some design advantages. You can create multiple event listeners that perform different actions when a sought file is found. Combining the different listeners can create more robust algorithms.

Here is the initial event argument declaration for finding a sought file:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Even though this type looks like a small, data-only type, you should follow the convention and make it a reference (`class`) type. That means the argument object will be passed by reference, and any updates to the data will be viewed by all subscribers. The first version is an immutable object. You should prefer to make the

properties in your event argument type immutable. That way, one subscriber cannot change the values before another subscriber sees them. (There are exceptions to this, as you'll see below.)

Next, we need to create the event declaration in the FileSearcher class. Leveraging the `EventHandler<T>` type means that you don't need to create yet another type definition. You simply use a generic specialization.

Let's fill out the FileSearcher class to search for files that match a pattern, and raise the correct event when a match is discovered.

```
public class FileSearcher
{
    public event EventHandler<FileEventArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileEventArgs(file));
        }
    }
}
```

Defining and Raising Field-Like Events

The simplest way to add an event to your class is to declare that event as a public field, as in the preceding example:

```
public event EventHandler<FileEventArgs> FileFound;
```

This looks like it's declaring a public field, which would appear to be bad object-oriented practice. You want to protect data access through properties, or methods. While this may look like a bad practice, the code generated by the compiler does create wrappers so that the event objects can only be accessed in safe ways. The only operations available on a field-like event are add handler:

```
EventHandler<FileEventArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

and remove handler:

```
fileLister.FileFound -= onFileFound;
```

Note that there's a local variable for the handler. If you used the body of the lambda, the remove would not work correctly. It would be a different instance of the delegate, and silently do nothing.

Code outside the class cannot raise the event, nor can it perform any other operations.

Returning Values from Event Subscribers

Your simple version is working fine. Let's add another feature: Cancellation.

When you raise the found event, listeners should be able to stop further processing, if this file is that last one

sought.

The event handlers do not return a value, so you need to communicate that in another way. The standard event pattern uses the EventArgs object to include fields that event subscribers can use to communicate cancel.

There are two different patterns that could be used, based on the semantics of the Cancel contract. In both cases, you'll add a boolean field to the EventArguments for the found file event.

One pattern would allow any one subscriber to cancel the operation. For this pattern, the new field is initialized to `false`. Any subscriber can change it to `true`. After all subscribers have seen the event raised, the FileSearcher component examines the boolean value and takes action.

The second pattern would only cancel the operation if all subscribers wanted the operation cancelled. In this pattern, the new field is initialized to indicate the operation should cancel, and any subscriber could change it to indicate the operation should continue. After all subscribers have seen the event raised, the FileSearcher component examines the boolean and takes action. There is one extra step in this pattern: the component needs to know if any subscribers have seen the event. If there are no subscribers, the field would indicate a cancel incorrectly.

Let's implement the first version for this sample. You need to add a boolean field named `CancelRequested` to the `FileFoundArgs` type:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

This new Field is automatically initialized to `false`, the default value for a Boolean field, so you don't cancel accidentally. The only other change to the component is to check the flag after raising the event to see if any of the subscribers have requested a cancellation:

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

One advantage of this pattern is that it isn't a breaking change. None of the subscribers requested a cancellation before, and they still are not. None of the subscriber code needs updating unless they want to support the new cancel protocol. It's very loosely coupled.

Let's update the subscriber so that it requests a cancellation once it finds the first executable:

```

EventHandler<FileEventArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};

```

Adding Another Event Declaration

Let's add one more feature, and demonstrate other language idioms for events. Let's add an overload of the `Search` method that traverses all subdirectories in search of files.

This could get to be a lengthy operation in a directory with many sub-directories. Let's add an event that gets raised when each new directory search begins. This enables subscribers to track progress, and update the user as to progress. All the samples you've created so far are public. Let's make this one an internal event. That means you can also make the types used for the arguments internal as well.

You'll start by creating the new `EventArgs` derived class for reporting the new directory and progress.

```

internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}

```

Again, you can follow the recommendations to make an immutable reference type for the event arguments.

Next, define the event. This time, you'll use a different syntax. In addition to using the field syntax, you can explicitly create the property, with `add` and `remove` handlers. In this sample, you won't need extra code in those handlers, but this shows how you would create them.

```

internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { directoryChanged += value; }
    remove { directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs> directoryChanged;

```

In many ways, the code you write here mirrors the code the compiler generates for the field event definitions you've seen earlier. You create the event using syntax very similar to that used for [properties](#). Notice that the handlers have different names: `add` and `remove`. These are called to subscribe to the event, or unsubscribe from the event. Notice that you also must declare a private backing field to store the event variable. It is initialized to null.

Next, let's add the overload of the `Search` method that traverses subdirectories and raises both events. The easiest way to accomplish this is to use a default argument to specify that you want to search all directories:

```

public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}

```

At this point, you can run the application calling the overload for searching all sub-directories. There are no subscribers on the new `ChangeDirectory` event, but using the `??.Invoke()` idiom ensures that this works correctly.

Let's add a handler to write a line that shows the progress in the console window.

```

fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.WriteLine($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($"{eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed...");
};

```

You've seen patterns that are followed throughout the .NET ecosystem. By learning these patterns and conventions, you'll be writing idiomatic C# and .NET quickly.

Next, you'll see some changes in these patterns in the most recent release of .NET.

[Next](#)

The Updated .NET Core Event Pattern

3/12/2020 • 3 minutes to read • [Edit Online](#)

[Previous](#)

The previous article discussed the most common event patterns. .NET Core has a more relaxed pattern. In this version, the `EventHandler<TEventArgs>` definition no longer has the constraint that `TEventArgs` must be a class derived from `System.EventArgs`.

This increases flexibility for you, and is backwards compatible. Let's start with the flexibility. The class `System.EventArgs` introduces one method: `MemberwiseClone()`, which creates a shallow copy of the object. That method must use reflection in order to implement its functionality for any class derived from `EventArgs`. That functionality is easier to create in a specific derived class. That effectively means that deriving from `System.EventArgs` is a constraint that limits your designs, but does not provide any additional benefit. In fact, you can change the definitions of `FileEventArgs` and `SearchDirectoryEventArgs` so that they do not derive from `EventArgs`. The program will work exactly the same.

You could also change the `SearchDirectoryEventArgs` to a struct, if you make one more change:

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

The additional change is to call the parameterless constructor before entering the constructor that initializes all the fields. Without that addition, the rules of C# would report that the properties are being accessed before they have been assigned.

You should not change the `FileEventArgs` from a class (reference type) to a struct (value type). That's because the protocol for handling cancel requires that the event arguments are passed by reference. If you made the same change, the file search class could never observe any changes made by any of the event subscribers. A new copy of the structure would be used for each subscriber, and that copy would be a different copy than the one seen by the file search object.

Next, let's consider how this change can be backwards compatible. The removal of the constraint does not affect any existing code. Any existing event argument types do still derive from `System.EventArgs`. Backwards compatibility is one major reason why they will continue to derive from `System.EventArgs`. Any existing event subscribers will be subscribers to an event that followed the classic pattern.

Following similar logic, any event argument type created now would not have any subscribers in any existing codebases. New event types that do not derive from `System.EventArgs` will not break those codebases.

Events with Async subscribers

You have one final pattern to learn: How to correctly write event subscribers that call async code. The challenge is described in the article on [async and await](#). Async methods can have a void return type, but that is strongly discouraged. When your event subscriber code calls an async method, you have no choice but to create an `async void` method. The event handler signature requires it.

You need to reconcile this opposing guidance. Somehow, you must create a safe `async void` method. The basics of the pattern you need to implement are below:

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

First, notice that the handler is marked as an async handler. Because it is being assigned to an event handler delegate type, it will have a void return type. That means you must follow the pattern shown in the handler, and not allow any exceptions to be thrown out of the context of the async handler. Because it does not return a task, there is no task that can report the error by entering the faulted state. Because the method is async, the method can't simply throw the exception. (The calling method has continued execution because it is `async`.) The actual runtime behavior will be defined differently for different environments. It may terminate the thread or the process that owns the thread, or leave the process in an indeterminate state. All of these potential outcomes are highly undesirable.

That's why you should wrap the `await` statement for the async Task in your own try block. If it does cause a faulted task, you can log the error. If it is an error from which your application cannot recover, you can exit the program quickly and gracefully.

Those are the major updates to the .NET event pattern. You will see many examples of the earlier versions in the libraries you work with. However, you should understand what the latest patterns are as well.

The next article in this series helps you distinguish between using `delegates` and `events` in your designs. They are similar concepts, and that article will help you make the best decision for your programs.

[Next](#)

Distinguishing Delegates and Events

11/2/2020 • 3 minutes to read • [Edit Online](#)

[Previous](#)

Developers that are new to the .NET Core platform often struggle when deciding between a design based on `delegates` and a design based on `events`. The choice of delegates or events is often difficult, because the two language features are similar. Events are even built using the language support for delegates.

They both offer a late binding scenario: they enable scenarios where a component communicates by calling a method that is only known at runtime. They both support single and multiple subscriber methods. You may find this referred to as singlecast and multicast support. They both support similar syntax for adding and removing handlers. Finally, raising an event and calling a delegate use exactly the same method call syntax. They even both support the same `Invoke()` method syntax for use with the `?.` operator.

With all those similarities, it is easy to have trouble determining when to use which.

Listening to Events is Optional

The most important consideration in determining which language feature to use is whether or not there must be an attached subscriber. If your code must call the code supplied by the subscriber, you should use a design based on delegates when you need to implement callback. If your code can complete all its work without calling any subscribers, you should use a design based on events.

Consider the examples built during this section. The code you built using `List.Sort()` must be given a comparer function in order to properly sort the elements. LINQ queries must be supplied with delegates in order to determine what elements to return. Both used a design built with delegates.

Consider the `Progress` event. It reports progress on a task. The task continues to proceed whether or not there are any listeners. The `FileSearcher` is another example. It would still search and find all the files that were sought, even with no event subscribers attached. UX controls still work correctly, even when there are no subscribers listening to the events. They both use designs based on events.

Return Values Require Delegates

Another consideration is the method prototype you would want for your delegate method. As you've seen, the delegates used for events all have a void return type. You've also seen that there are idioms to create event handlers that do pass information back to event sources through modifying properties of the event argument object. While these idioms do work, they are not as natural as returning a value from a method.

Notice that these two heuristics may often both be present: If your delegate method returns a value, it will likely impact the algorithm in some way.

Events Have Private Invocation

Classes other than the one in which an event is contained can only add and remove event listeners; only the class containing the event can invoke the event. Events are typically public class members. By comparison, delegates are often passed as parameters and stored as private class members, if they are stored at all.

Event Listeners Often Have Longer Lifetimes

That event listeners have longer lifetimes is a slightly weaker justification. However, you may find that event-

based designs are more natural when the event source will be raising events over a long period of time. You can see examples of event-based design for UX controls on many systems. Once you subscribe to an event, the event source may raise events throughout the lifetime of the program. (You can unsubscribe from events when you no longer need them.)

Contrast that with many delegate-based designs, where a delegate is used as an argument to a method, and the delegate is not used after that method returns.

Evaluate Carefully

The above considerations are not hard and fast rules. Instead, they represent guidance that can help you decide which choice is best for your particular usage. Because they are similar, you can even prototype both, and consider which would be more natural to work with. They both handle late binding scenarios well. Use the one that communicates your design the best.

Language Integrated Query (LINQ)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see [Type relationships in LINQ query operations](#).
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see [Introduction to LINQ queries](#).

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see [C# language specification](#) and [Standard query operators overview](#).
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as [Count](#) or [Max](#), have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query syntax and method syntax in LINQ](#).
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<T>](#) queries are compiled to delegates. [IQueryable](#) and [IQueryable<T>](#) queries are compiled to expression trees. For more information, see [Expression trees](#).

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in [Query expression basics](#), and then read the documentation for the LINQ technology in which you are interested:

- XML documents: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to entities](#)
- .NET collections, files, strings and so on: [LINQ to objects](#)

To gain a deeper understanding of LINQ in general, see [LINQ in C#](#).

To start working with LINQ in C#, see the tutorial [Working with LINQ](#).

Query expression basics

3/12/2020 • 12 minutes to read • [Edit Online](#)

This article introduces the basic concepts related to query expressions in C#.

What is a query and what does it do?

A *query* is a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have. A query is distinct from the results that it produces.

Generally, the source data is organized logically as a sequence of elements of the same kind. For example, a SQL database table contains a sequence of rows. In an XML file, there is a "sequence" of XML elements (although these are organized hierarchically in a tree structure). An in-memory collection contains a sequence of objects.

From an application's viewpoint, the specific type and structure of the original source data is not important. The application always sees the source data as an `IEnumerable<T>` or `IQueryable<T>` collection. For example, in LINQ to XML, the source data is made visible as an `IEnumerable< XElement >`.

Given this source sequence, a query may do one of three things:

- Retrieve a subset of the elements to produce a new sequence without modifying the individual elements. The query may then sort or group the returned sequence in various ways, as shown in the following example (assume `scores` is an `int[]`):

```
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
```

- Retrieve a sequence of elements as in the previous example but transform them to a new type of object. For example, a query may retrieve only the last names from certain customer records in a data source. Or it may retrieve the complete record and then use it to construct another in-memory object type or even XML data before generating the final result sequence. The following example shows a projection from an `int` to a `string`. Note the new type of `highScoresQuery`.

```
IEnumerable<string> highScoresQuery2 =
    from score in scores
    where score > 80
    orderby score descending
    select $"The score is {score}";
```

- Retrieve a singleton value about the source data, such as:
 - The number of elements that match a certain condition.
 - The element that has the greatest or least value.
 - The first element that matches a condition, or the sum of particular values in a specified set of elements. For example, the following query returns the number of scores greater than 80 from the `scores` integer array:

```
int highScoreCount =  
    (from score in scores  
     where score > 80  
     select score)  
.Count();
```

In the previous example, note the use of parentheses around the query expression before the call to the `Count` method. You can also express this by using a new variable to store the concrete result. This technique is more readable because it keeps the variable that stores the query separate from the query that stores a result.

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
int scoreCount = highScoresQuery3.Count();
```

In the previous example, the query is executed in the call to `Count`, because `Count` must iterate over the results in order to determine the number of elements returned by `highScoresQuery`.

What is a query expression?

A *query expression* is a query expressed in query syntax. A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid. A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery. Each clause in turn contains one or more C# expressions, and these expressions may themselves be either a query expression or contain a query expression.

A query expression must begin with a `from` clause and must end with a `select` or `group` clause. Between the first `from` clause and the last `select` or `group` clause, it can contain one or more of these optional clauses: `where`, `orderby`, `join`, `let` and even additional `from` clauses. You can also use the `into` keyword to enable the result of a `join` or `group` clause to serve as the source for additional query clauses in the same query expression.

Query variable

In LINQ, a query variable is any variable that stores a *query* instead of the *results* of a query. More specifically, a query variable is always an enumerable type that will produce a sequence of elements when it is iterated over in a `foreach` statement or a direct call to its `IEnumerator.MoveNext` method.

The following code example shows a simple query expression with one data source, one filtering clause, one ordering clause, and no transformation of the source elements. The `select` clause ends the query.

```

static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82

```

In the previous example, `scoreQuery` is a *query variable*, which is sometimes referred to as just a *query*. The query variable stores no actual result data, which is produced in the `foreach` loop. And when the `foreach` statement executes, the query results are not returned through the query variable `scoreQuery`. Rather, they are returned through the iteration variable `testScore`. The `scoreQuery` variable can be iterated in a second `foreach` loop. It will produce the same results as long as neither it nor the data source has been modified.

A query variable may store a query that is expressed in query syntax or method syntax, or a combination of the two. In the following examples, both `queryMajorCities` and `queryMajorCities2` are query variables:

```

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);

```

On the other hand, the following two examples show variables that are not query variables even though each is initialized with a query. They are not query variables because they store results:

```

int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
    .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();

```

For more information about the different ways to express queries, see [Query syntax and method syntax in LINQ](#).

Explicit and implicit typing of query variables

This documentation usually provides the explicit type of the query variable in order to show the type relationship between the query variable and the [select clause](#). However, you can also use the `var` keyword to instruct the compiler to infer the type of a query variable (or any other local variable) at compile time. For example, the query example that was shown previously in this topic can also be expressed by using implicit typing:

```

// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 10000
    select city;

```

For more information, see [Implicitly typed local variables](#) and [Type relationships in LINQ query operations](#).

Starting a query expression

A query expression must begin with a `from` clause. It specifies a data source together with a range variable. The range variable represents each successive element in the source sequence as the source sequence is being traversed. The range variable is strongly typed based on the type of elements in the data source. In the following example, because `countries` is an array of `Country` objects, the range variable is also typed as `country`.

Because the range variable is strongly typed, you can use the dot operator to access any available members of the type.

```
IEnumerable<Country> countryAreaQuery =  
    from country in countries  
    where country.Area > 500000 //sq km  
    select country;
```

The range variable is in scope until the query is exited either with a semicolon or with a *continuation* clause.

A query expression may contain multiple `from` clauses. Use additional `from` clauses when each element in the source sequence is itself a collection or contains a collection. For example, assume that you have a collection of `Country` objects, each of which contains a collection of `City` objects named `Cities`. To query the `City` objects in each `Country`, use two `from` clauses as shown here:

```
IEnumerable<City> cityQuery =  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city;
```

For more information, see [from clause](#).

Ending a query expression

A query expression must end with either a `group` clause or a `select` clause.

group clause

Use the `group` clause to produce a sequence of groups organized by a key that you specify. The key can be any data type. For example, the following query creates a sequence of groups that contains one or more `Country` objects and whose key is a `char` value.

```
var queryCountryGroups =  
    from country in countries  
    group country by country.Name[0];
```

For more information about grouping, see [group clause](#).

select clause

Use the `select` clause to produce all other types of sequences. A simple `select` clause just produces a sequence of the same type of objects as the objects that are contained in the data source. In this example, the data source contains `Country` objects. The `orderby` clause just sorts the elements into a new order and the `select` clause produces a sequence of the reordered `Country` objects.

```
IEnumerable<Country> sortedQuery =  
    from country in countries  
    orderby country.Area  
    select country;
```

The `select` clause can be used to transform source data into sequences of new types. This transformation is also named a *projection*. In the following example, the `select` clause *projects* a sequence of anonymous types which contains only a subset of the fields in the original element. Note that the new objects are initialized by using an object initializer.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

For more information about all the ways that a `select` clause can be used to transform source data, see [select clause](#).

Continuations with "into"

You can use the `into` keyword in a `select` or `group` clause to create a temporary identifier that stores a query. Do this when you must perform additional query operations on a query after a grouping or select operation. In the following example `countries` are grouped according to population in ranges of 10 million. After these groups are created, additional clauses filter out some groups, and then to sort the groups in ascending order. To perform those additional operations, the continuation represented by `countryGroup` is required.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

For more information, see [into](#).

Filtering, ordering, and joining

Between the starting `from` clause, and the ending `select` or `group` clause, all other clauses (`where`, `join`, `orderby`, `from`, `let`) are optional. Any of the optional clauses may be used zero times or multiple times in a query body.

`where` clause

Use the `where` clause to filter out elements from the source data based on one or more predicate expressions. The `where` clause in the following example has one predicate with two conditions.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

For more information, see [where clause](#).

`orderby` clause

Use the `orderby` clause to sort the results in either ascending or descending order. You can also specify secondary sort orders. The following example performs a primary sort on the `country` objects by using the `Area` property. It then performs a secondary sort by using the `Population` property.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

The `ascending` keyword is optional; it is the default sort order if no order is specified. For more information, see [orderby clause](#).

join clause

Use the `join` clause to associate and/or combine elements from one data source with elements from another data source based on an equality comparison between specified keys in each element. In LINQ, join operations are performed on sequences of objects whose elements are different types. After you have joined two sequences, you must use a `select` or `group` statement to specify which element to store in the output sequence. You can also use an anonymous type to combine properties from each set of associated elements into a new type for the output sequence. The following example associates `prod` objects whose `Category` property matches one of the categories in the `categories` string array. Products whose `Category` does not match any string in `categories` are filtered out. The `select` statement projects a new type whose properties are taken from both `cat` and `prod`.

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

You can also perform a group join by storing the results of the `join` operation into a temporary variable by using the `into` keyword. For more information, see [join clause](#).

let clause

Use the `let` clause to store the result of an expression, such as a method call, in a new range variable. In the following example, the range variable `firstName` stores the first element of the array of strings that is returned by `Split`.

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.WriteLine(s + " ");
//Output: Svetlana Claire Sven Cesar
```

For more information, see [let clause](#).

Subqueries in a query expression

A query clause may itself contain a query expression, which is sometimes referred to as a *subquery*. Each subquery starts with its own `from` clause that does not necessarily point to the same data source in the first `from` clause. For example, the following query shows a query expression that is used in the `select` statement to retrieve the results of a grouping operation.

```
var queryGroupMax =  
    from student in students  
    group student by student.GradeLevel into studentGroup  
    select new  
    {  
        Level = studentGroup.Key,  
        HighestScore =  
            (from student2 in studentGroup  
             select student2.Scores.Average())  
            .Max()  
    };
```

For more information, see [Perform a subquery on a grouping operation](#).

See also

- [C# programming guide](#)
- [Language Integrated Query \(LINQ\)](#)
- [Query keywords \(LINQ\)](#)
- [Standard query operators overview](#)

LINQ in C#

11/1/2019 • 2 minutes to read • [Edit Online](#)

This section contains links to topics that provide more detailed information about LINQ.

In this section

[Introduction to LINQ queries](#)

Describes the three parts of the basic LINQ query operation that are common across all languages and data sources.

[LINQ and generic types](#)

Provides a brief introduction to generic types as they are used in LINQ.

[Data transformations with LINQ](#)

Describes the various ways that you can transform data retrieved in queries.

[Type relationships in LINQ query operations](#)

Describes how types are preserved and/or transformed in the three parts of a LINQ query operation

[Query syntax and method syntax in LINQ](#)

Compares method syntax and query syntax as two ways to express a LINQ query.

[C# features that support LINQ](#)

Describes the language constructs in C# that support LINQ.

Related sections

[LINQ query expressions](#)

Includes an overview of queries in LINQ and provides links to additional resources.

[Standard query operators overview](#)

Introduces the standard methods used in LINQ.

Write LINQ queries in C#

11/2/2020 • 4 minutes to read • [Edit Online](#)

This article shows the three ways in which you can write a LINQ query in C#:

1. Use query syntax.
2. Use method syntax.
3. Use a combination of query syntax and method syntax.

The following examples demonstrate some simple LINQ queries by using each approach listed previously. In general, the rule is to use (1) whenever possible, and use (2) and (3) whenever necessary.

NOTE

These queries operate on simple in-memory collections; however, the basic syntax is identical to that used in LINQ to Entities and LINQ to XML.

Example - Query syntax

The recommended way to write most queries is to use *query syntax* to create *query expressions*. The following example shows three query expressions. The first query expression demonstrates how to filter or restrict results by applying conditions with a `where` clause. It returns all elements in the source sequence whose values are greater than 7 or less than 3. The second expression demonstrates how to order the returned results. The third expression demonstrates how to group results according to a key. This query returns two groups based on the first letter of the word.

```
// Query #1.  
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
  
// The query variable can also be implicitly typed by using var  
IQueryable<int> filteringQuery =  
    from num in numbers  
    where num < 3 || num > 7  
    select num;  
  
// Query #2.  
IQueryable<int> orderingQuery =  
    from num in numbers  
    where num < 3 || num > 7  
    orderby num ascending  
    select num;  
  
// Query #3.  
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };  
IEnumerable<IGrouping<char, string>> queryFoodGroups =  
    from item in groupingQuery  
    group item by item[0];
```

Note that the type of the queries is `IEnumerable<T>`. All of these queries could be written using `var` as shown in the following example:

```
var query = from num in numbers...
```

In each previous example, the queries do not actually execute until you iterate over the query variable in a `foreach` statement or other statement. For more information, see [Introduction to LINQ Queries](#).

Example - Method syntax

Some query operations must be expressed as a method call. The most common such methods are those that return singleton numeric values, such as [Sum](#), [Max](#), [Min](#), [Average](#), and so on. These methods must always be called last in any query because they represent only a single value and cannot serve as the source for an additional query operation. The following example shows a method call in a query expression:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IQueryable<int> concatenationQuery = numbers1.Concat(numbers2);
```

If the method has Action or Func parameters, these are provided in the form of a [lambda](#) expression, as shown in the following example:

```
// Query #6.
IQueryable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

In the previous queries, only Query #4 executes immediately. This is because it returns a single value, and not a generic `IEnumerable<T>` collection. The method itself has to use `foreach` in order to compute its value.

Each of the previous queries can be written by using implicit typing with `var`, as shown in the following example:

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Example - Mixed query and method syntax

This example shows how to use method syntax on the results of a query clause. Just enclose the query expression in parentheses, and then apply the dot operator and call the method. In the following example, query #7 returns a count of the numbers whose value is between 3 and 7. In general, however, it is better to use a second variable to store the result of the method call. In this manner, the query is less likely to be confused with the results of the query.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Because Query #7 returns a single value and not a collection, the query executes immediately.

The previous query can be written by using implicit typing with `var`, as follows:

```
var numCount = (from num in numbers...
```

It can be written in method syntax as follows:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

It can be written by using explicit typing, as follows:

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

See also

- [Walkthrough: Writing Queries in C#](#)
- [Language Integrated Query \(LINQ\)](#)
- [where clause](#)

Query a collection of objects

1/23/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to perform a simple query over a list of `Student` objects. Each `Student` object contains some basic information about the student, and a list that represents the student's scores on four examinations.

This application serves as the framework for many other examples in this section that use the same `students` data source.

Example

The following query returns the students who received a score of 90 or greater on their first exam.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79 }},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94 }},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87 }},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82 }},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70 }},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78 }},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91 }},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39 }},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60 }},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92 }},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91 }},
    };
}
```

```

        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
                     Year = GradeLevel.FourthYear,
                     ExamScores = new List<int> { 96, 85, 91, 60}}
    };
#endregion

// Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public static void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                     where student.ExamScores[exam] > score
                     select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

This query is intentionally simple to enable you to experiment. For example, you can try more conditions in the `where` clause, or use an `orderby` clause to sort the results.

See also

- [Language Integrated Query \(LINQ\)](#)
- [String interpolation](#)

How to return a query from a method (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to return a query from a method as the return value and as an `out` parameter.

Query objects are composable, meaning that you can return a query from a method. Objects that represent queries do not store the resulting collection, but rather the steps to produce the results when needed. The advantage of returning query objects from methods is that they can be further composed or modified. Therefore any return value or `out` parameter of a method that returns a query must also have that type. If a method materializes a query into a concrete `List<T>` or `Array` type, it is considered to be returning the query results instead of the query itself. A query variable that is returned from a method can still be composed or modified.

Example

In the following example, the first method returns a query as its value, and the second method returns a query as an `out` parameter. Note that in both cases it is a query that is returned, not query results.

```
class MQ
{
    // QueryMethod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // ...
    }
}
```

```

// directly, without using myQuery1.
Console.WriteLine("\nResults of executing myQuery1 directly:");
// Rest the mouse pointer over the call to QueryMethod1 to see its
// return type.
foreach (string s in app.QueryMethod1(ref nums))
{
    Console.WriteLine(s);
}

IEnumerable<string> myQuery2;
// QueryMethod2 returns a query as the value of its out parameter.
app.QueryMethod2(ref nums, out myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (string s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. A saved query
// is nested inside a new query definition that revises the results
// of the first query.
myQuery1 = from item in myQuery1
            orderby item descending
            select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (string s in myQuery1)
{
    Console.WriteLine(s);
}

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}

```

See also

- [Language Integrated Query \(LINQ\)](#)

Store the results of a query in memory

5/15/2019 • 2 minutes to read • [Edit Online](#)

A query is basically a set of instructions for how to retrieve and organize data. Queries are executed lazily, as each subsequent item in the result is requested. When you use `foreach` to iterate the results, items are returned as accessed. To evaluate a query and store its results without executing a `foreach` loop, just call one of the following methods on the query variable:

- [ToList](#)
- [ToArray](#)
- [ToDictionary](#)
- [ToLookup](#)

We recommend that when you store the query results, you assign the returned collection object to a new variable as shown in the following example:

Example

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

See also

- [Language Integrated Query \(LINQ\)](#)

Group query results

1/24/2019 • 8 minutes to read • [Edit Online](#)

Grouping is one of the most powerful capabilities of LINQ. The following examples show how to group data in various ways:

- By a single property.
- By the first letter of a string property.
- By a computed numeric range.
- By Boolean predicate or other expression.
- By a compound key.

In addition, the last two queries project their results into a new anonymous type that contains only the student's first and last name. For more information, see the [group clause](#).

Example

All the examples in this topic use the following helper classes and data sources.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 96, 87, 80, 84}}
    }
}
```

```

        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 75, 84, 91, 39}),
    new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
        Year = GradeLevel.SecondYear,
        ExamScores = new List<int>{ 97, 92, 81, 60}),
    new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
        Year = GradeLevel.ThirdYear,
        ExamScores = new List<int>{ 68, 79, 88, 92}),
    new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
        Year = GradeLevel.FirstYear,
        ExamScores = new List<int>{ 94, 92, 91, 91}),
    new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 96, 85, 91, 60})
};

#endregion

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                    where student.ExamScores[exam] > score
                    select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

Example

The following example shows how to group source elements by using a single property of the element as the group key. In this case the key is a `string`, the student's last name. It is also possible to use a substring for the key. The grouping operation uses the default equality comparer for the type.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySingleProperty()`.

```

public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by a single property in an object:
Key: Adams
    Adams, Terry
Key: Fakhouri
    Fakhouri, Fadi
Key: Feng
    Feng, Hanying
Key: Garcia
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: Mortensen
    Mortensen, Sven
Key: O'Donnell
    O'Donnell, Claire
Key: Omelchenko
    Omelchenko, Svetlana
Key: Tucker
    Tucker, Lance
    Tucker, Michael
Key: Zabokritski
    Zabokritski, Eugene
*/

```

Example

The following example shows how to group source elements by using something other than a property of the object for the group key. In this example, the key is the first letter of the student's last name.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySubstring()`.

```

public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by something other than a property of the object:
Key: A
    Adams, Terry
Key: F
    Fakhouri, Fadi
    Feng, Hanying
Key: G
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: M
    Mortensen, Sven
Key: O
    O'Donnell, Claire
    Omelchenko, Svetlana
Key: T
    Tucker, Lance
    Tucker, Michael
Key: Z
    Zabokritski, Eugene
*/

```

Example

The following example shows how to group source elements by using a numeric range as a group key. The query then projects the results into an anonymous type that contains only the first and last name and the percentile range to which the student belongs. An anonymous type is used because it is not necessary to use the complete `Student` object to display the results. `GetPercentile` is a helper function that calculates a percentile based on the student's average score. The method returns an integer between 0 and 10.

```

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

```

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByRange()`.

```

public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"{item.LastName}, {item.FirstName}");
        }
    }
}

/* Output:
Group by numeric range and project into a new anonymous type:
Key: 60
    Garcia, Debra
Key: 70
    O'Donnell, Claire
Key: 80
    Adams, Terry
    Feng, Hanying
    Garcia, Cesar
    Garcia, Hugo
    Mortensen, Sven
    Omelchenko, Svetlana
    Tucker, Lance
    Zabokritski, Eugene
Key: 90
    Fakhouri, Fadi
    Tucker, Michael
*/

```

Example

The following example shows how to group source elements by using a Boolean comparison expression. In this example, the Boolean expression tests whether a student's average exam score is greater than 75. As in previous examples, the results are projected into an anonymous type because the complete source element is not needed. Note that the properties in the anonymous type become properties on the `key` member and can be accessed by name when the query is executed.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByBoolean()`.

```

public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                                group new { student.FirstName, student.LastName } 
                                by student.ExamScores.Average() > 75 into studentGroup
                                select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
/* Output:
Group by a Boolean into two groups with string keys
"True" and "False" and project into a new anonymous type:
Key: True
    Terry Adams
    Fadi Fakhouri
    Hanying Feng
    Cesar Garcia
    Hugo Garcia
    Sven Mortensen
    Svetlana Omelchenko
    Lance Tucker
    Michael Tucker
    Eugene Zabokritski
Key: False
    Debra Garcia
    Claire O'Donnell
*/

```

Example

The following example shows how to use an anonymous type to encapsulate a key that contains multiple values. In this example, the first key value is the first letter of the student's last name. The second key value is a Boolean that specifies whether the student scored over 85 on the first exam. You can order the groups by any property in the key.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByCompositeKey()`.

```

public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
Group and order by a compound key:
Name starts with A who scored more than 85
    Terry Adams
Name starts with F who scored more than 85
    Fadi Fakhouri
    Hanying Feng
Name starts with G who scored more than 85
    Cesar Garcia
    Hugo Garcia
Name starts with G who scored less than 85
    Debra Garcia
Name starts with M who scored more than 85
    Sven Mortensen
Name starts with O who scored less than 85
    Claire O'Donnell
Name starts with O who scored more than 85
    Svetlana Omelchenko
Name starts with T who scored less than 85
    Lance Tucker
Name starts with T who scored more than 85
    Michael Tucker
Name starts with Z who scored more than 85
    Eugene Zabokritski
*/

```

See also

- [GroupBy](#)
- [IGrouping<TKey,TElement>](#)
- [Language Integrated Query \(LINQ\)](#)
- [group clause](#)
- [Anonymous Types](#)
- [Perform a Subquery on a Grouping Operation](#)
- [Create a Nested Group](#)
- [Grouping Data](#)

Create a nested group

1/24/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to create nested groups in a LINQ query expression. Each group that is created according to student year or grade level is then further subdivided into groups based on the individuals' names.

Example

NOTE

This example contains references to objects that are defined in the sample code in [Query a collection of objects](#).

```

public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
            group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"Names that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}

/*
Output:
DataClass.Student Level = SecondYear
    Names that begin with: Adams
        Adams Terry
    Names that begin with: Garcia
        Garcia Hugo
    Names that begin with: Omelchenko
        Omelchenko Svetlana
DataClass.Student Level = ThirdYear
    Names that begin with: Fakhouri
        Fakhouri Fadi
    Names that begin with: Garcia
        Garcia Debra
    Names that begin with: Tucker
        Tucker Lance
DataClass.Student Level = FirstYear
    Names that begin with: Feng
        Feng Hanying
    Names that begin with: Mortensen
        Mortensen Sven
    Names that begin with: Tucker
        Tucker Michael
DataClass.Student Level = FourthYear
    Names that begin with: Garcia
        Garcia Cesar
    Names that begin with: O'Donnell
        O'Donnell Claire
    Names that begin with: Zabokritski
        Zabokritski Eugene
*/

```

Note that three nested `foreach` loops are required to iterate over the inner elements of a nested group.

See also

- [Language Integrated Query \(LINQ\)](#)

Perform a subquery on a grouping operation

3/12/2020 • 2 minutes to read • [Edit Online](#)

This article shows two different ways to create a query that orders the source data into groups, and then performs a subquery over each group individually. The basic technique in each example is to group the source elements by using a *continuation* named `newGroup`, and then generating a new subquery against `newGroup`. This subquery is run against each new group that is created by the outer query. Note that in this particular example the final output is not a group, but a flat sequence of anonymous types.

For more information about how to group, see [group clause](#).

For more information about continuations, see [into](#). The following example uses an in-memory data structure as the data source, but the same principles apply for any kind of LINQ data source.

Example

NOTE

This example contains references to objects that are defined in the sample code in [Query a collection of objects](#).

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
                (from student2 in studentGroup
                 select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
}
```

The query in the snippet above can also be written using method syntax. The following code snippet has a semantically equivalent query written using method syntax.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
    {
        Level = studentGroup.Key,
        HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
    });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
}
```

See also

- [Language Integrated Query \(LINQ\)](#)

Group results by contiguous keys

8/24/2018 • 7 minutes to read • [Edit Online](#)

The following example shows how to group elements into chunks that represent subsequences of contiguous keys. For example, assume that you are given the following sequence of key-value pairs:

KEY	VALUE
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

The following groups will be created in this order:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

The solution is implemented as an extension method that is thread-safe and that returns its results in a streaming manner. In other words, it produces its groups as it moves through the source sequence. Unlike the `group` or `orderby` operators, it can begin returning groups to the caller before all of the sequence has been read.

Thread-safety is accomplished by making a copy of each group or chunk as the source sequence is iterated, as explained in the source code comments. If the source sequence has a large sequence of contiguous items, the common language runtime may throw an [OutOfMemoryException](#).

Example

The following example shows both the extension method and the client code that uses it:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```

namespace ChunkIT
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IGrouping<TKey, TSource> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
                // current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
                // ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
                // elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
                // more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the
                iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source
        // sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            // (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)
        }
    }
}

```

```

// A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk.
Each ChunkItem
    // has a reference to the next ChunkItem in the list.
    class ChunkItem
    {
        public ChunkItem(TSource value)
        {
            Value = value;
        }
        public readonly TSource Value;
        public ChunkItem Next = null;
    }

    // The value that is used to determine matching elements
    private readonly TKey key;

    // Stores a reference to the enumerator for the source sequence
    private Ienumerator<TSource> enumerator;

    // A reference to the predicate that is used to compare keys.
    private Func<TSource, bool> predicate;

    // Stores the contents of the first source element that
    // belongs with this chunk.
    private readonly ChunkItem head;

    // End of the list. It is repositioned each time a new
    // ChunkItem is added.
    private ChunkItem tail;

    // Flag to indicate the source iterator has reached the end of the source sequence.
    internal bool isLastSourceElement = false;

    // Private object for thread synchronization
    private object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, Ienumerator<TSource> enumerator, Func<TSource, bool> predicate)
    {
        this.key = key;
        this.enumerator = enumerator;
        this.predicate = predicate;

        // A Chunk always contains at least one element.
        head = new ChunkItem(enumerator.Current);

        // The end and beginning are the same until the list contains > 1 elements.
        tail = head;

        m_Lock = new object();
    }

    // Indicates that all chunk elements have been copied to the list of ChunkItems,
    // and the source enumerator is either at the end, or else on an element with a new key.
    // the tail of the linked list is set to null in the CopyNextChunkElement method if the
    // key of the next element does not match the current chunk's key, or there are no more elements
    in the source.
    private bool DoneCopyingChunk => tail == null;

    // Adds one ChunkItem to the current group
    // REQUIRES: !DoneCopyingChunk && lock(this)
    private void CopyNextChunkElement()
    {
        // Try to advance the iterator on the source sequence.
        // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
        isLastSourceElement = !enumerator.MoveNext();

        // If we are (a) at the end of the source, or (b) at the end of the current chunk

```

```

        // then null out the enumerator and predicate for reuse with the next chunk.
        if (isLastSourceElement || !predicate(enumerator.Current))
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next;
    }

    // Called after the end of the last chunk was reached. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key => key;

    // Invoked by the inner foreach loop. This method stays just one step ahead
    // of the client requests. It adds the next element of the chunk only after
    // the clients requests the last element in the list so far.
    public IEnumrator<TSource> GetEnumerator()
    {
        //Specify the initial element to enumerate.
        ChunkItem current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }
}

```

```

        }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
GetEnumerator();
    }
}

// A simple named type is used for easier viewing in the debugger. Anonymous types
// work just as well with the ChunkBy operator.
public class KeyValuePair
{
    public string Key { get; set; }
    public string Value { get; set; }
}

class Program
{
    // The source sequence.
    public static IEnumerable<KeyValuePair> list;

    // Query variable declared as class member to be available
    // on different threads.
    static IGrouping<string, KeyValuePair>> query;

    static void Main(string[] args)
    {
        // Initialize the source sequence with an array initializer.
        list = new[]
        {
            new KeyValuePair{ Key = "A", Value = "We" },
            new KeyValuePair{ Key = "A", Value = "think" },
            new KeyValuePair{ Key = "A", Value = "that" },
            new KeyValuePair{ Key = "B", Value = "Linq" },
            new KeyValuePair{ Key = "C", Value = "is" },
            new KeyValuePair{ Key = "A", Value = "really" },
            new KeyValuePair{ Key = "B", Value = "cool" },
            new KeyValuePair{ Key = "B", Value = "!" }
        };
        // Create the query by using our user-defined query operator.
        query = list.ChunkBy(p => p.Key);

        // ChunkBy returns IGrouping objects, therefore a nested
        // foreach loop is required to access the elements in each "chunk".
        foreach (var item in query)
        {
            Console.WriteLine($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"{inner.Value}");
            }
        }

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
}

```

To use the extension method in your project, copy the `MyExtensions` static class to a new or existing source code file and if it is required, add a `using` directive for the namespace where it is located.

See also

- [Language Integrated Query \(LINQ\)](#)

Dynamically specify predicate filters at runtime

1/24/2019 • 2 minutes to read • [Edit Online](#)

In some cases, you don't know until run time how many predicates you have to apply to source elements in the `where` clause. One way to dynamically specify multiple predicate filters is to use the `Contains` method, as shown in the following example. The example is constructed in two ways. First, the project is run by filtering on values that are provided in the program. Then the project is run again by using input provided at run time.

To filter by using the Contains method

1. Open a new console application and name it `PredicateFilters`.
2. Copy the `StudentClass` class from [Query a collection of objects](#) and paste it into namespace `PredicateFilters` underneath class `Program`. `StudentClass` provides a list of `Student` objects.
3. Comment out the `Main` method in `StudentClass`.
4. Replace class `Program` with the following code:

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Add the following line to the `Main` method in class `DynamicPredicates`, under the declaration of `ids`.

```
QueryById(ids);
```

6. Run the project.

7. The following output is displayed in a console window:

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. The next step is to run the project again, this time by using input entered at run time instead of array `ids`. Change `QueryByID(ids)` to `QueryByID(args)` in the `Main` method.
9. Run the project with the command line arguments `122 117 120 115`. When the project is run, those values become elements of `args`, the parameter of the `Main` method.
10. The following output is displayed in a console window:

Adams: 120

Feng: 117

Garcia: 115

Tucker: 122

To filter by using a switch statement

1. You can use a `switch` statement to select among predetermined alternative queries. In the following example, `studentQuery` uses a different `where` clause depending on which grade level, or year, is specified at run time.
2. Copy the following method and paste it into class `DynamicPredicates`.

```

// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;
        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

```

3. In the `Main` method, replace the call to `QueryByID` with the following call, which sends the first element from the `args` array as its argument: `QueryByYear(args[0])`.
4. Run the project with a command line argument of an integer value between 1 and 4.

See also

- [Language Integrated Query \(LINQ\)](#)
- [where clause](#)

Perform inner joins

1/24/2019 • 10 minutes to read • [Edit Online](#)

In relational database terms, an *inner join* produces a result set in which each element of the first collection appears one time for every matching element in the second collection. If an element in the first collection has no matching elements, it does not appear in the result set. The `Join` method, which is called by the `join` clause in C#, implements an inner join.

This article shows you how to perform four variations of an inner join:

- A simple inner join that correlates elements from two data sources based on a simple key.
- An inner join that correlates elements from two data sources based on a *composite* key. A composite key, which is a key that consists of more than one value, enables you to correlate elements based on more than one property.
- A *multiple join* in which successive join operations are appended to each other.
- An inner join that is implemented by using a group join.

Example - Simple key join

The following example creates two collections that contain objects of two user-defined types, `Person` and `Pet`. The query uses the `join` clause in C# to match `Person` objects with `Pet` objects whose `Owner` is that `Person`. The `select` clause in C# defines how the resulting objects will look. In this example the resulting objects are anonymous types that consist of the owner's first name and the pet's name.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"{ownerAndPet.PetName} is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui

```

Note that the `Person` object whose `Lastname` is "Huff" does not appear in the result set because there is no `Pet` object that has `Pet.Owner` equal to that `Person`.

Example - Composite key join

Instead of correlating elements based on just one property, you can use a composite key to compare elements based on multiple properties. To do this, specify the key selector function for each collection to return an anonymous type that consists of the properties you want to compare. If you label the properties, they must have the same label in each key's anonymous type. The properties must also appear in the same order.

The following example uses a list of `Employee` objects and a list of `Student` objects to determine which

employees are also students. Both of these types have a `FirstName` and a `LastName` property of type `String`. The functions that create the join keys from each list's elements return an anonymous type that consists of the `FirstName` and `LastName` properties of each element. The join operation compares these composite keys for equality and returns pairs of objects from each list where both the first name and the last name match.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
        new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
        new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
        new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Example - Multiple join

Any number of join operations can be appended to each other to perform a multiple join. Each `join` clause in C# correlates a specified data source with the results of the previous join.

The following example creates three collections: a list of `Person` objects, a list of `Cat` objects, and a list of `Dog` objects.

The first `join` clause in C# matches people and cats based on a `Person` object matching `Cat.Owner`. It returns a sequence of anonymous types that contain the `Person` object and `Cat.Name`.

The second `join` clause in C# correlates the anonymous types returned by the first join with `Dog` objects in the supplied list of dogs, based on a composite key that consists of the `Owner` property of type `Person`, and the first letter of the animal's name. It returns a sequence of anonymous types that contain the `Cat.Name` and `Dog.Name` properties from each matching pair. Because this is an inner join, only those objects from the first data source that have a match in the second data source are returned.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
               join cat in cats on person equals cat.Owner
               join dog in dogs on
                   new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                   equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
               select new { CatName = cat.Name, DogName = dog.Name };
```

```

foreach (var obj in query)
{
    Console.WriteLine(
        $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"{obj.DogName}\".");
}
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".

```

Example - Inner join by using grouped join

The following example shows you how to implement an inner join by using a group join.

In `query1`, the list of `Person` objects is group-joined to the list of `Pet` objects based on the `Person` matching the `Pet.Owner` property. The group join creates a collection of intermediate groups, where each group consists of a `Person` object and a sequence of matching `Pet` objects.

By adding a second `from` clause to the query, this sequence of sequences is combined (or flattened) into one longer sequence. The type of the elements of the final sequence is specified by the `select` clause. In this example, that type is an anonymous type that consists of the `Person.FirstName` and `Pet.Name` properties for each matching pair.

The result of `query1` is equivalent to the result set that would have been obtained by using the `join` clause without the `into` clause to perform an inner join. The `query2` variable demonstrates this equivalent query.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people

```

```

        join pet in pets on person equals pet.Owner into gj
        from subpet in gj
        select new { OwnerName = person.FirstName, PetName = subpet.Name };

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

var query2 = from person in people
            join pet in pets on person equals pet.Owner
            select new { OwnerName = person.FirstName, PetName = pet.Name };

Console.WriteLine("\nThe equivalent operation using Join():");
foreach (var v in query2)
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers

```

See also

- [Join](#)
- [GroupJoin](#)
- [Perform grouped joins](#)
- [Perform left outer joins](#)
- [Anonymous types](#)

Perform grouped joins

11/2/2020 • 5 minutes to read • [Edit Online](#)

The group join is useful for producing hierarchical data structures. It pairs each element from the first collection with a set of correlated elements from the second collection.

For example, a class or a relational database table named `Student` might contain two fields: `Id` and `Name`. A second class or relational database table named `Course` might contain two fields: `StudentId` and `CourseTitle`. A group join of these two data sources, based on matching `Student.Id` and `Course.StudentId`, would group each `Student` with a collection of `course` objects (which might be empty).

NOTE

Each element of the first collection appears in the result set of a group join regardless of whether correlated elements are found in the second collection. In the case where no correlated elements are found, the sequence of correlated elements for that element is empty. The result selector therefore has access to every element of the first collection. This differs from the result selector in a non-group join, which cannot access elements from the first collection that have no match in the second collection.

WARNING

`Enumerable.GroupJoin` has no direct equivalent in traditional relational database terms. However, this method does implement a superset of inner joins and left outer joins. Both of these operations can be written in terms of a grouped join. For more information, see [Join Operations](#) and [Entity Framework Core, GroupJoin](#).

The first example in this article shows you how to perform a group join. The second example shows you how to use a group join to create XML elements.

Example - Group join

The following example performs a group join of objects of type `Person` and `Pet` based on the `Person` matching the `Pet.Owner` property. Unlike a non-group join, which would produce a pair of elements for each match, the group join produces only one resulting object for each element of the first collection, which in this example is a `Person` object. The corresponding elements from the second collection, which in this example are `Pet` objects, are grouped into a collection. Finally, the result selector function creates an anonymous type for each match that consists of `Person.FirstName` and a collection of `Pet` objects.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
        join pet in pets on person equals pet.Owner into gj
        select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:

```

Example - Group join to create XML

Group joins are ideal for creating XML by using LINQ to XML. The following example is similar to the previous example except that instead of creating anonymous types, the result selector function creates XML elements that

represent the joined objects.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));
}

Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

See also

- [Join](#)
- [GroupJoin](#)
- [Perform inner joins](#)
- [Perform left outer joins](#)
- [Anonymous types](#)

Perform left outer joins

1/24/2019 • 2 minutes to read • [Edit Online](#)

A left outer join is a join in which each element of the first collection is returned, regardless of whether it has any correlated elements in the second collection. You can use LINQ to perform a left outer join by calling the [DefaultIfEmpty](#) method on the results of a group join.

Example

The following example demonstrates how to use the [DefaultIfEmpty](#) method on the results of a group join to perform a left outer join.

The first step in producing a left outer join of two collections is to perform an inner join by using a group join. (See [Perform inner joins](#) for an explanation of this process.) In this example, the list of `Person` objects is inner-joined to the list of `Pet` objects based on a `Person` object that matches `Pet.Owner`.

The second step is to include each element of the first (left) collection in the result set even if that element has no matches in the right collection. This is accomplished by calling [DefaultIfEmpty](#) on each sequence of matching elements from the group join. In this example, [DefaultIfEmpty](#) is called on each sequence of matching `Pet` objects. The method returns a collection that contains a single, default value if the sequence of matching `Pet` objects is empty for any `Person` object, thereby ensuring that each `Person` object is represented in the result collection.

NOTE

The default value for a reference type is `null`; therefore, the example checks for a null reference before accessing each element of each `Pet` collection.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName} : {v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:      Daisy
// Terry:       Barley
// Terry:       Boots
// Terry:       Blue Moon
// Charlotte:   Whiskers
// Arlene:

```

See also

- [Join](#)
- [GroupJoin](#)
- [Perform inner joins](#)
- [Perform grouped joins](#)
- [Anonymous types](#)

Order the results of a join clause

1/24/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to order the results of a join operation. Note that the ordering is performed after the join. Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we do not recommend it. Some LINQ providers might not preserve that ordering after the join.

Example

This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a sub-query orders all the matching elements from the products sequence.

```
class HowToOrderJoins
{
    #region Data
    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()

    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
        new Category() { Name="Grains", ID=004},
        new Category() { Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()

    {
        new Product{Name="Cola", CategoryID=001},
        new Product{Name="Tea", CategoryID=001},
        new Product{Name="Mustard", CategoryID=002},
        new Product{Name="Pickles", CategoryID=002},
        new Product{Name="Carrots", CategoryID=003},
        new Product{Name="Bok Choy", CategoryID=003},
        new Product{Name="Peaches", CategoryID=005},
        new Product{Name="Melons", CategoryID=005},
    };

    #endregion
    static void Main()
    {
        HowToOrderJoins app = new HowToOrderJoins();
        app.OrderJoin1();

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```

void OrderJoin1()
{
    var groupJoinQuery2 =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        orderby category.Name
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                        orderby prod2.Name
                        select prod2
        };

    foreach (var productGroup in groupJoinQuery2)
    {
        Console.WriteLine(productGroup.Category);
        foreach (var prodItem in productGroup.Products)
        {
            Console.WriteLine($" {prodItem.Name,-10} {prodItem.CategoryID}");
        }
    }
}
/* Output:
   Beverages
      Cola      1
      Tea       1
   Condiments
      Mustard   2
      Pickles   2
   Fruit
      Melons    5
      Peaches   5
   Grains
   Vegetables
      Bok Choy  3
      Carrots   3
*/
}

```

See also

- [Language Integrated Query \(LINQ\)](#)
- [orderby clause](#)
- [join clause](#)

Join by using composite keys

1/24/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to perform join operations in which you want to use more than one key to define a match. This is accomplished by using a composite key. You create a composite key as an anonymous type or named typed with the values that you want to compare. If the query variable will be passed across method boundaries, use a named type that overrides [Equals](#) and [GetHashCode](#) for the key. The names of the properties, and the order in which they occur, must be identical in each key.

Example

The following example demonstrates how to use a composite key to join data from three tables:

```
var query = from o in db.Orders
            from p in db.Products
            join d in db.OrderDetails
                on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
                from d in details
            select new {o.OrderID, p.ProductID, d.UnitPrice};
```

Type inference on composite keys depends on the names of the properties in the keys, and the order in which they occur. If the properties in the source sequences don't have the same names, you must assign new names in the keys. For example, if the `Orders` table and `OrderDetails` table each used different names for their columns, you could create composite keys by assigning identical names in the anonymous types:

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Composite keys can be also used in a `group` clause.

See also

- [Language Integrated Query \(LINQ\)](#)
- [join clause](#)
- [group clause](#)

Perform custom join operations

1/24/2019 • 5 minutes to read • [Edit Online](#)

This example shows how to perform join operations that aren't possible with the `join` clause. In a query expression, the `join` clause is limited to, and optimized for, equijoins, which are by far the most common type of join operation. When performing an equijoin, you will probably always get the best performance by using the `join` clause.

However, the `join` clause cannot be used in the following cases:

- When the join is predicated on an expression of inequality (a non-equijoin).
- When the join is predicated on more than one expression of equality or inequality.
- When you have to introduce a temporary range variable for the right side (inner) sequence before the join operation.

To perform joins that aren't equijoins, you can use multiple `from` clauses to introduce each data source independently. You then apply a predicate expression in a `where` clause to the range variable for each source. The expression also can take the form of a method call.

NOTE

Don't confuse this kind of custom join operation with the use of multiple `from` clauses to access inner collections. For more information, see [join clause](#).

Example

The first method in the following example shows a simple cross join. Cross joins must be used with caution because they can produce very large result sets. However, they can be useful in some scenarios for creating source sequences against which additional queries are run.

The second method produces a sequence of all the products whose category ID is listed in the category list on the left side. Note the use of the `let` clause and the `Contains` method to create a temporary array. It also is possible to create the array before the query and eliminate the first `from` clause.

```
class CustomJoins
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
```

```

{
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
};

// Specify the second data source.
List<Product> products = new List<Product>()
{
    new Product{Name="Tea", CategoryID=001},
    new Product{Name="Mustard", CategoryID=002},
    new Product{Name="Pickles", CategoryID=002},
    new Product{Name="Carrots", CategoryID=003},
    new Product{Name="Bok Choy", CategoryID=003},
    new Product{Name="Peaches", CategoryID=005},
    new Product{Name="Melons", CategoryID=005},
    new Product{Name="Ice Cream", CategoryID=007},
    new Product{Name="Mackerel", CategoryID=012},
};
#endifregion

static void Main()
{
    CustomJoins app = new CustomJoins();
    app.CrossJoin();
    app.NonEquijoin();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

void CrossJoin()
{
    var crossJoinQuery =
        from c in categories
        from p in products
        select new { c.ID, p.Name };

    Console.WriteLine("Cross Join Query:");
    foreach (var v in crossJoinQuery)
    {
        Console.WriteLine($"{v.ID},-{5}{v.Name}");
    }
}

void NonEquijoin()
{
    var nonEquijoinQuery =
        from p in products
        let catIds = from c in categories
                    select c.ID
        where catIds.Contains(p.CategoryID) == true
        select new { Product = p.Name, CategoryID = p.CategoryID };

    Console.WriteLine("Non-equijoin query:");
    foreach (var v in nonEquijoinQuery)
    {
        Console.WriteLine($"{v.CategoryID},-{5}{v.Product}");
    }
}

/* Output:
Cross Join Query:
1   Tea
1   Mustard
1   Pickles
1   Carrots
1   Bok Choy
1   Peaches

```

```

1   Melons
1   Ice Cream
1   Mackerel
2   Tea
2   Mustard
2   Pickles
2   Carrots
2   Bok Choy
2   Peaches
2   Melons
2   Ice Cream
2   Mackerel
3   Tea
3   Mustard
3   Pickles
3   Carrots
3   Bok Choy
3   Peaches
3   Melons
3   Ice Cream
3   Mackerel
Non-equijoin query:
1   Tea
2   Mustard
2   Pickles
3   Carrots
3   Bok Choy
Press any key to exit.
 */

```

Example

In the following example, the query must join two sequences based on matching keys that, in the case of the inner (right side) sequence, cannot be obtained prior to the join clause itself. If this join were performed with a `join` clause, then the `Split` method would have to be called for each element. The use of multiple `from` clauses enables the query to avoid the overhead of the repeated method call. However, since `join` is optimized, in this particular case it might still be faster than using multiple `from` clauses. The results will vary depending primarily on how expensive the method call is.

```

class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)

```

```

        select Convert.ToInt32(scoreAsText)).
        ToList()
    };

    // Optional. Store the newly created student objects in memory
    // for faster access in future queries
    List<Student> students = queryNamesScores.ToList();

    foreach (var student in students)
    {
        Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
{student.ExamScores.Average()}.");
    }

    //Keep console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/

```

See also

- [Language Integrated Query \(LINQ\)](#)
- [join clause](#)
- [Order the results of a join clause](#)

Handle null values in query expressions

3/25/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to handle possible null values in source collections. An object collection such as an `IEnumerable<T>` can contain elements whose value is `null`. If a source collection is null or contains an element whose value is null, and your query does not handle null values, a `NullReferenceException` will be thrown when you execute the query.

Example

You can code defensively to avoid a null reference exception as shown in the following example:

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

In the previous example, the `where` clause filters out all null elements in the `categories` sequence. This technique is independent of the null check in the join clause. The conditional expression with `null` in this example works because `Products.CategoryID` is of type `int?` which is shorthand for `Nullable<int>`.

Example

In a join clause, if only one of the comparison keys is a nullable value type, you can cast the other to a nullable value type in the query expression. In the following example, assume that `EmployeeID` is a column that contains values of type `int?`:

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

See also

- [Nullable<T>](#)
- [Language Integrated Query \(LINQ\)](#)
- [Nullable value types](#)

Handle exceptions in query expressions

1/24/2019 • 2 minutes to read • [Edit Online](#)

It's possible to call any method in the context of a query expression. However, we recommend that you avoid calling any method in a query expression that can create a side effect such as modifying the contents of the data source or throwing an exception. This example shows how to avoid raising exceptions when you call methods in a query expression without violating the general .NET guidelines on exception handling. Those guidelines state that it's acceptable to catch a specific exception when you understand why it's thrown in a given context. For more information, see [Best Practices for Exceptions](#).

The final example shows how to handle those cases when you must throw an exception during execution of a query.

Example

The following example shows how to move exception handling code outside a query expression. This is only possible when the method does not depend on any variables local to the query.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

Example

In some cases, the best response to an exception that is thrown from within a query might be to stop the query execution immediately. The following example shows how to handle exceptions that might be thrown from inside a query body. Assume that `SomeMethodThatMightThrow` can potentially cause an exception that requires the query execution to stop.

Note that the `try` block encloses the `foreach` loop, and not the query itself. This is because the `foreach` loop is the point at which the query is actually executed. For more information, see [Introduction to LINQ queries](#).

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
   Processing C:\newFolder\fileA.txt
   Processing C:\newFolder\fileB.txt
   Operation is not valid due to the current state of the object.
*/
```

See also

- [Language Integrated Query \(LINQ\)](#)

Pattern Matching

11/2/2020 • 13 minutes to read • [Edit Online](#)

Patterns test that a value has a certain *shape*, and can *extract* information from the value when it has the matching shape. Pattern matching provides more concise syntax for algorithms you already use today. You already create pattern matching algorithms using existing syntax. You write `if` or `switch` statements that test values. Then, when those statements match, you extract and use information from that value. The new syntax elements are extensions to statements you're already familiar with: `is` and `switch`. These new extensions combine testing a value and extracting that information.

In this article, we'll look at the new syntax to show you how it enables readable, concise code. Pattern matching enables idioms where data and the code are separated, unlike object-oriented designs where data and the methods that manipulate them are tightly coupled.

To illustrate these new idioms, let's work with structures that represent geometric shapes using pattern matching statements. You're probably familiar with building class hierarchies and creating [virtual methods](#) and [overridden methods](#) to customize object behavior based on the runtime type of the object.

Those techniques aren't possible for data that isn't structured in a class hierarchy. When data and methods are separate, you need other tools. The new *pattern matching* constructs enable cleaner syntax to examine data and manipulate control flow based on any condition of that data. You already write `if` statements and `switch` that test a variable's value. You write `is` statements that test a variable's type. *Pattern matching* adds new capabilities to those statements.

In this article, you'll build a method that computes the area of different geometric shapes. But, you'll do it without resorting to object-oriented techniques and building a class hierarchy for the different shapes. You'll use *pattern matching* instead. As you go through this sample, contrast this code with how it would be structured as an object hierarchy. When the data you must query and manipulate isn't a class hierarchy, pattern matching enables elegant designs.

Rather than starting with an abstract shape definition and adding different specific shape classes, let's start instead with simple data only definitions for each of the geometric shapes:

```
public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}

public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
}

public struct Rectangle
{
    public double Length { get; }
    public double Height { get; }

    public Rectangle(double length, double height)
    {
        Length = length;
        Height = height;
    }
}

public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}
```

From these structures, let's write a method that computes the area of some shape.

The `is` type pattern expression

Before C# 7.0, you'd need to test each type in a series of `if` and `is` statements:

```

public static double ComputeArea(object shape)
{
    if (shape is Square)
    {
        var s = (Square)shape;
        return s.Side * s.Side;
    }
    else if (shape is Circle)
    {
        var c = (Circle)shape;
        return c.Radius * c.Radius * Math.PI;
    }
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}

```

That code above is a classic expression of the *type pattern*: You're testing a variable to determine its type and taking a different action based on that type.

This code becomes simpler using extensions to the `is` expression to assign a variable if the test succeeds:

```

public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}

```

In this updated version, the `is` expression both tests the variable and assigns it to a new variable of the proper type. Also, notice that this version includes the `Rectangle` type, which is a `struct`. The new `is` expression works with value types as well as reference types.

Language rules for pattern matching expressions help you avoid misusing the results of a match expression. In the example above, the variables `s`, `c`, and `r` are only in scope and definitely assigned when the respective pattern match expressions have `true` results. If you try to use either variable in another location, your code generates compiler errors.

Let's examine both of those rules in detail, beginning with scope. The variable `c` is in scope only in the `else` branch of the first `if` statement. The variable `s` is in scope in the method `ComputeAreaModernIs`. That's because each branch of an `if` statement establishes a separate scope for variables. However, the `if` statement itself doesn't. That means variables declared in the `if` statement are in the same scope as the `if` statement (the method in this case). This behavior isn't specific to pattern matching, but is the defined behavior for variable scopes and `if` and `else` statements.

The variables `c` and `s` are assigned when the respective `if` statements are true because of the definitely assigned when true mechanism.

TIP

The samples in this topic use the recommended construct where a pattern match `is` expression definitely assigns the match variable in the `true` branch of the `if` statement. You could reverse the logic by saying `if (!(shape is Square s))` and the variable `s` would be definitely assigned only in the `false` branch. While this is valid C#, it is not recommended because it is more confusing to follow the logic.

These rules mean that you're unlikely to accidentally access the result of a pattern match expression when that pattern wasn't met.

Using pattern matching `switch` statements

As time goes on, you may need to support other shape types. As the number of conditions you're testing grows, you'll find that using the `is` pattern matching expressions can become cumbersome. In addition to requiring `if` statements on each type you want to check, the `is` expressions are limited to testing if the input matches a single type. In this case, you'll find that the `switch` pattern matching expressions becomes a better choice.

The traditional `switch` statement was a pattern expression: it supported the constant pattern. You could compare a variable to any constant used in a `case` statement:

```
public static string GenerateMessage(params string[] parts)
{
    switch (parts.Length)
    {
        case 0:
            return "No elements to the input";
        case 1:
            return $"One element: {parts[0]}";
        case 2:
            return $"Two elements: {parts[0]}, {parts[1]}";
        default:
            return $"Many elements. Too many to write";
    }
}
```

The only pattern supported by the `switch` statement was the constant pattern. It was further limited to numeric types and the `string` type. Those restrictions have been removed, and you can now write a `switch` statement using the type pattern:

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
            return r.Height * r.Length;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

The pattern matching `switch` statement uses familiar syntax to developers who have used the traditional C-

style `switch` statement. Each `case` is evaluated and the code beneath the condition that matches the input variable is executed. Code execution can't "fall through" from one case expression to the next; the syntax of the `case` statement requires that each `case` end with a `break`, `return`, or `goto`.

NOTE

The `goto` statements to jump to another label are valid only for the constant pattern (the classic switch statement).

There are important new rules governing the `switch` statement. The restrictions on the type of the variable in the `switch` expression have been removed. Any type, such as `object` in this example, may be used. The case expressions are no longer limited to constant values. Removing that limitation means that reordering `switch` sections may change a program's behavior.

When limited to constant values, no more than one `case` label could match the value of the `switch` expression. Combine that with the rule that every `switch` section must not fall through to the next section, and it followed that the `switch` sections could be rearranged in any order without affecting behavior. Now, with more generalized `switch` expressions, the order of each section matters. The `switch` expressions are evaluated in textual order. Execution transfers to the first `switch` label that matches the `switch` expression. The `default` case will only be executed if no other case labels match. The `default` case is evaluated last, regardless of its textual order. If there's no `default` case, and none of the other `case` statements match, execution continues at the statement following the `switch` statement. None of the `case` labels code is executed.

when clauses in case expressions

You can make special cases for those shapes that have 0 area by using a `when` clause on the `case` label. A square with a side length of 0, or a circle with a radius of 0 has a 0 area. You specify that condition using a `when` clause on the `case` label:

```
public static double ComputeArea_Version3(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

This change demonstrates a few important points about the new syntax. First, multiple `case` labels can be applied to one `switch` section. The statement block is executed when any of those labels is `true`. In this instance, if the `switch` expression is either a circle or a square with 0 area, the method returns the constant 0.

This example introduces two different variables in the two `case` labels for the first `switch` block. Notice that the statements in this `switch` block don't use either the variables `c` (for the circle) or `s` (for the square). Neither of those variables is definitely assigned in this `switch` block. If either of these cases match, clearly one

of the variables has been assigned. However, it's impossible to tell *which* has been assigned at compile time, because either case could match at runtime. For that reason, most times when you use multiple `case` labels for the same block, you won't introduce a new variable in the `case` statement, or you'll only use the variable in the `when` clause.

Having added those shapes with 0 area, let's add a couple more shape types: a rectangle and a triangle:

```
public static double ComputeArea_Version4(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

This set of changes adds `case` labels for the degenerate case, and labels and blocks for each of the new shapes.

Finally, you can add a `null` case to ensure the argument isn't `null`:

```
public static double ComputeArea_Version5(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        case null:
            throw new ArgumentNullException(paramName: nameof(shape), message: "Shape must not be null");
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

The special behavior for the `null` pattern is interesting because the constant `null` in the pattern doesn't have a type but can be converted to any reference type or nullable value type. Rather than convert a `null` to any type, the language defines that a `null` value won't match any type pattern, regardless of the compile-time type of the variable. This behavior makes the new `switch` based type pattern consistent with the `is` statement: `is` statements always return `false` when the value being checked is `null`. It's also simpler: once you've checked the type, you don't need an additional null check. You can see that from the fact that there are no null checks in any of the case blocks of the samples above: they aren't necessary, since matching the type pattern guarantees a non-null value.

`var` declarations in `case` expressions

The introduction of `var` as one of the match expressions introduces new rules to the pattern match.

The first rule is that the `var` declaration follows the normal type inference rules: The type is inferred to be the static type of the switch expression. From that rule, the type always matches.

The second rule is that a `var` declaration doesn't have the null check that other type pattern expressions include. That means the variable may be null, and a null check is necessary in that case.

Those two rules mean that in many instances, a `var` declaration in a `case` expression matches the same conditions as a `default` expression. Because any non-default case is preferred to the `default` case, the `default` case will never execute.

NOTE

The compiler does not emit a warning in those cases where a `default` case has been written but will never execute. This is consistent with current `switch` statement behavior where all possible cases have been listed.

The third rule introduces uses where a `var` case may be useful. Imagine that you're doing a pattern match where the input is a string and you're searching for known command values. You might write something like:

```
static object CreateShape(string shapeDescription)
{
    switch (shapeDescription)
    {
        case "circle":
            return new Circle(2);

        case "square":
            return new Square(4);

        case "large-circle":
            return new Circle(12);

        case var o when (o?.Trim().Length ?? 0) == 0:
            // white space
            return null;
        default:
            return "invalid shape description";
    }
}
```

The `var` case matches `null`, the empty string, or any string that contains only white space. Notice that the preceding code uses the `?.` operator to ensure that it doesn't accidentally throw a `NullReferenceException`. The `default` case handles any other string values that aren't understood by this command parser.

This is one example where you may want to consider a `var` case expression that is distinct from a `default`

expression.

Conclusions

Pattern Matching constructs enable you to easily manage control flow among different variables and types that aren't related by an inheritance hierarchy. You can also control logic to use any condition you test on the variable. It enables patterns and idioms that you'll need more often as you build more distributed applications, where data and the methods that manipulate that data are separate. You'll notice that the shape structs used in this sample don't contain any methods, just read-only properties. Pattern Matching works with any data type. You write expressions that examine the object, and make control flow decisions based on those conditions.

Compare the code from this sample with the design that would follow from creating a class hierarchy for an abstract `Shape` and specific derived shapes each with their own implementation of a virtual method to calculate the area. You'll often find that pattern matching expressions can be a very useful tool when you're working with data and want to separate the data storage concerns from the behavior concerns.

See also

- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)

Write safe and efficient C# code

3/23/2021 • 16 minutes to read • [Edit Online](#)

New features in C# enable you to write verifiable safe code with better performance. If you carefully apply these techniques, fewer scenarios require unsafe code. These features make it easier to use references to value types as method arguments and method returns. When done safely, these techniques minimize copying value types. By using value types, you can minimize the number of allocations and garbage collection passes.

Much of the sample code in this article uses features added in C# 7.2. To use those features, you must configure your project to use C# 7.2 or later. For more information on setting the language version, see [configure the language version](#).

This article focuses on techniques for efficient resource management. One advantage to using value types is that they often avoid heap allocations. The disadvantage is that they're copied by value. This trade-off makes it harder to optimize algorithms that operate on large amounts of data. New language features in C# 7.2 provide mechanisms that enable safe efficient code using references to value types. Use these features wisely to minimize both allocations and copy operations. This article explores those new features.

This article focuses on the following resource management techniques:

- Declare a `readonly struct` to express that a type is **immutable**. That enables the compiler to save defensive copies when using `in` parameters.
- If a type can't be immutable, declare `struct` members `readonly` to indicate that the member doesn't modify state.
- Use a `ref readonly` return when the return value is a `struct` larger than `IntPtrSize` and the storage lifetime is greater than the method returning the value.
- When the size of a `readonly struct` is bigger than `IntPtrSize`, you should pass it as an `in` parameter for performance reasons.
- Never pass a `struct` as an `in` parameter unless it's declared with the `readonly` modifier or the method calls only `readonly` members of the struct. Violating this guidance may negatively affect performance and could lead to an obscure behavior.
- Use a `ref struct`, or a `readonly ref struct` such as `Span<T>` or `ReadOnlySpan<T>` to work with memory as a sequence of bytes.

These techniques force you to balance two competing goals with regard to **references** and **values**. Variables that are **reference types** hold a reference to the location in memory. Variables that are **value types** directly contain their value. These differences highlight the key differences that are important for managing memory resources. **Value types** are typically copied when passed to a method or returned from a method. This behavior includes copying the value of `this` when calling members of a value type. The cost of the copy is related to the size of the type. **Reference types** are allocated on the managed heap. Each new object requires a new allocation, and subsequently must be reclaimed. Both these operations take time. The reference is copied when a reference type is passed as an argument to a method or returned from a method.

This article uses the following example concept of the 3D-point structure to explain these recommendations:

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

Different examples use different implementations of this concept.

Declare readonly structs for immutable value types

Declaring a `struct` using the `readonly` modifier informs the compiler that your intent is to create an immutable type. The compiler enforces that design decision with the following rules:

- All field members must be `readonly`
- All properties must be read-only, including auto-implemented properties.

These two rules are sufficient to ensure that no member of a `readonly struct` modifies the state of that struct.

The `struct` is immutable. The `Point3D` structure could be defined as an immutable struct as shown in the following example:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Follow this recommendation whenever your design intent is to create an immutable value type. Any performance improvements are an added benefit. The `readonly struct` clearly expresses your design intent.

Declare readonly members when a struct can't be immutable

In C# 8.0 and later, when a struct type is mutable, you should declare members that don't cause mutation to be `readonly`. Consider a different application that needs a 3D point structure, but must support mutability. The following version of the 3D point structure adds the `readonly` modifier only to those members that don't modify the structure. Follow this example when your design must support modifications to the struct by some members, but you still want the benefits of enforcing `readonly` on some members:

```

public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

    public readonly double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);

    public readonly override string ToString() => $"{X}, {Y}, {Z}";
}

```

The preceding sample shows many of the locations where you can apply the `readonly` modifier: methods, properties, and property accessors. If you use auto-implemented properties, the compiler adds the `readonly` modifier to the `get` accessor for read-write properties. The compiler adds the `readonly` modifier to the auto-implemented property declarations for properties with only a `get` accessor.

Adding the `readonly` modifier to members that don't mutate state provides two related benefits. First, the compiler enforces your intent. That member can't mutate the struct's state. Second, the compiler won't create defensive copies of `in` parameters when accessing a `readonly` member. The compiler can make this optimization safely because it guarantees that the `struct` is not modified by a `readonly` member.

Use `ref readonly return` statements for large structures when possible

You can return values by reference when the value being returned isn't local to the returning method. Returning by reference means that only the reference is copied, not the structure. In the following example, the `Origin` property can't use a `ref` return because the value being returned is a local variable:

```
public Point3D Origin => new Point3D(0,0,0);
```

However, the following property definition can be returned by reference because the returned value is a static member:

```

public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}

```

You don't want callers modifying the origin, so you should return the value by `ref readonly`:

```

public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}

```

Returning `ref readonly` enables you to save copying larger structures and preserve the immutability of your internal data members.

At the call site, callers make the choice to use the `Origin` property as a `ref readonly` or as a value:

```

var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;

```

The first assignment in the preceding code makes a copy of the `origin` constant and assigns that copy. The second assigns a reference. Notice that the `readonly` modifier must be part of the declaration of the variable. The reference to which it refers can't be modified. Attempts to do so result in a compile-time error.

The `readonly` modifier is required on the declaration of `originReference`.

The compiler enforces that the caller can't modify the reference. Attempts to assign the value directly generate a compile-time error. In other cases, the compiler allocates a defensive copy unless it can safely use the `readonly` reference. Static analysis rules determine if the struct could be modified. The compiler doesn't create a defensive copy when the struct is a `readonly struct` or the member is a `readonly` member of the struct. Defensive copies aren't needed to pass the struct as an `in` argument.

Apply the `in` modifier to `readonly struct` parameters larger than `System.IntPtr.Size`

The `in` keyword complements the existing `ref` and `out` keywords to pass arguments by reference. The `in` keyword specifies passing the argument by reference, but the called method doesn't modify the value.

This addition provides a full vocabulary to express your design intent. Value types are copied when passed to a called method when you don't specify any of the following modifiers in the method signature. Each of these modifiers specifies that a variable is passed by reference, avoiding the copy. Each modifier expresses a different intent:

- `out`: This method sets the value of the argument used as this parameter.
- `ref`: This method may set the value of the argument used as this parameter.
- `in`: This method doesn't modify the value of the argument used as this parameter.

Add the `in` modifier to pass an argument by reference and declare your design intent to pass arguments by reference to avoid unnecessary copying. You don't intend to modify the object used as that argument.

This practice often improves performance for readonly value types that are larger than `IntPtr.Size`. For simple types (`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` and `bool`, and `enum` types), any potential performance gains are minimal. In fact, performance may degrade by using pass-by-reference for types smaller than `IntPtr.Size`.

The following code shows an example of a method that calculates the distance between two points in 3D space.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The arguments are two structures that each contain three doubles. A double is 8 bytes, so each argument is 24 bytes. By specifying the `in` modifier, you pass a 4 byte or 8-byte reference to those arguments, depending on the architecture of the machine. The difference in size is small, but it adds up when your application calls this method in a tight loop using many different values.

The `in` modifier complements `out` and `ref` in other ways as well. You can't create overloads of a method that differ only in the presence of `in`, `out`, or `ref`. These new rules extend the same behavior that had always been defined for `out` and `ref` parameters. Like the `out` and `ref` modifiers, value types aren't boxed because the `in` modifier is applied.

The `in` modifier may be applied to any member that takes parameters: methods, delegates, lambdas, local functions, indexers, operators.

Another feature of `in` parameters is that you may use literal values or constants for the argument to an `in` parameter. Also, unlike a `ref` or `out` parameter, you don't need to apply the `in` modifier at the call site. The following code shows you two examples of calling the `CalculateDistance` method. The first uses two local variables passed by reference. The second includes a temporary variable created as part of the method call.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

There are several ways in which the compiler enforces the read-only nature of an `in` argument. First of all, the called method can't directly assign to an `in` parameter. It can't directly assign to any field of an `in` parameter when that value is a `struct` type. In addition, you can't pass an `in` parameter to any method using the `ref` or `out` modifier. These rules apply to any field of an `in` parameter, provided the field is a `struct` type and the parameter is also a `struct` type. In fact, these rules apply for multiple layers of member access provided the types at all levels of member access are `structs`. The compiler enforces that `struct` types passed as `in` arguments and their `struct` members are read-only variables when used as arguments to other methods.

The use of `in` parameters can avoid the potential performance costs of making copies. It doesn't change the semantics of any method call. Therefore, you don't need to specify the `in` modifier at the call site. Omitting the `in` modifier at the call site informs the compiler that it's allowed to make a copy of the argument for the following reasons:

- There exists an implicit conversion but not an identity conversion from the argument type to the parameter type.

- The argument is an expression but doesn't have a known storage variable.
- An overload exists that differs by the presence or absence of `in`. In that case, the `by value` overload is a better match.

These rules are useful as you update existing code to use read-only reference arguments. Inside the called method, you can call any instance method that uses `by value` parameters. In those instances, a copy of the `in` parameter is created. Because the compiler may create a temporary variable for any `in` parameter, you can also specify default values for any `in` parameter. The following code specifies the origin (point 0,0) as the default value for the second point:

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

To force the compiler to pass read-only arguments by reference, specify the `in` modifier on the arguments at the call site, as shown in the following code:

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

This behavior makes it easier to adopt `in` parameters over time in large codebases where performance gains are possible. You add the `in` modifier to method signatures first. Then, you can add the `in` modifier at call sites and create `readonly struct` types to enable the compiler to avoid creating defensive copies of `in` parameters in more locations.

The `in` parameter designation can also be used with reference types or numeric values. However, the benefits in both cases are minimal, if any.

Avoid mutable structs as an `in` argument

The techniques described above explain how to avoid copies by returning references and passing values by reference. These techniques work best when the argument types are declared as `readonly struct` types. Otherwise, the compiler must create **defensive copies** in many situations to enforce the `readonly`-ness of any arguments. Consider the following example that calculates the distance of a 3D point from the origin:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The `Point3D` structure is *not* a `readonly struct`. There are six different property access calls in the body of this method. On first examination, you may have thought these accesses were safe. After all, a `get` accessor shouldn't modify the state of the object. But there's no language rule that enforces that. It's only a common convention. Any type could implement a `get` accessor that modified the internal state. Without some language guarantee, the compiler must create a temporary copy of the argument before calling any member not marked

with the `readonly` modifier. The temporary storage is created on the stack, the values of the argument are copied to the temporary storage, and the value is copied to the stack for each member access as the `this` argument. In many situations, these copies harm performance enough that pass-by-value is faster than pass-by-readonly-reference when the argument type isn't a `readonly struct` and the method calls members that aren't marked `readonly`. If you mark all methods that don't modify the struct state as `readonly`, the compiler can safely determine that the struct state isn't modified, and a defensive copy is not needed.

Instead, if the distance calculation uses the immutable struct, `ReadOnlyPoint3D`, temporary objects aren't needed:

```
private static double CalculateDistance3(in ReadOnlyPoint3D point1, in ReadOnlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The compiler generates more efficient code when you call members of a `readonly struct`: The `this` reference, instead of a copy of the receiver, is always an `in` parameter passed by reference to the member method. This optimization saves copying when you use a `readonly struct` as an `in` argument.

You shouldn't pass a nullable value type as an `in` argument. The `Nullable<T>` type isn't declared as a read-only struct. That means the compiler must generate defensive copies for any nullable value type argument passed to a method using the `in` modifier on the parameter declaration.

You can see an example program that demonstrates the performance differences using [BenchmarkDotNet](#) in our [samples repository](#) on GitHub. It compares passing a mutable struct by value and by reference with passing an immutable struct by value and by reference. The use of the immutable struct and pass by reference is fastest.

Use `ref struct` types to work with blocks or memory on a single stack frame

A related language feature is the ability to declare a value type that must be constrained to a single stack frame. This restriction enables the compiler to make several optimizations. The primary motivation for this feature was `Span<T>` and related structures. You'll achieve performance improvements from these enhancements by using new and updated .NET APIs that make use of the `Span<T>` type.

You may have similar requirements working with memory created using `stackalloc` or when using memory from interop APIs. You can define your own `ref struct` types for those needs.

`readonly ref struct` type

Declaring a struct as `readonly ref` combines the benefits and restrictions of `ref struct` and `readonly struct` declarations. The memory used by the readonly span is restricted to a single stack frame, and the memory used by the readonly span can't be modified.

Conclusions

Using value types minimizes the number of allocation operations:

- Storage for value types is stack allocated for local variables and method arguments.
- Storage for value types that are members of other objects is allocated as part of that object, not as a separate allocation.
- Storage for value type return values is stack allocated.

Contrast that with reference types in those same situations:

- Storage for reference types are heap allocated for local variables and method arguments. The reference is stored on the stack.
- Storage for reference types that are members of other objects are separately allocated on the heap. The containing object stores the reference.
- Storage for reference type return values is heap allocated. The reference to that storage is stored on the stack.

Minimizing allocations comes with tradeoffs. You copy more memory when the size of the `struct` is larger than the size of a reference. A reference is typically 64 bits or 32 bits, and depends on the target machine CPU.

These tradeoffs generally have minimal performance impact. However, for large structs or larger collections, the performance impact increases. The impact can be large in tight loops and hot paths for programs.

These enhancements to the C# language are designed for performance critical algorithms where minimizing memory allocations is a major factor in achieving the necessary performance. You may find that you don't often use these features in the code you write. However, these enhancements have been adopted throughout .NET. As more and more APIs make use of these features, you'll see the performance of your applications improve.

See also

- [ref keyword](#)
- [Ref returns and ref locals](#)

Expression Trees

11/2/2020 • 2 minutes to read • [Edit Online](#)

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you are not familiar with LINQ, you probably want to read [the LINQ tutorial](#) and the article about [lambda expressions](#) before this one.) *Expression Trees* provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you can examine, modify, or execute. These tools give you the power to manipulate code during run time. You can write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you can modify running algorithms, and even translate C# expressions into another form for execution in another environment.

You've likely already written code that uses Expression Trees. Entity Framework's LINQ APIs accept Expression Trees as the arguments for the LINQ Query Expression Pattern. That enables [Entity Framework](#) to translate the query you wrote in C# into SQL that executes in the database engine. Another example is [Moq](#), which is a popular mocking framework for .NET.

The remaining sections of this tutorial will explore what Expression Trees are, examine the framework classes that support expression trees, and show you how to work with expression trees. You'll learn how to read expression trees, how to create expression trees, how to create modified expression trees, and how to execute the code represented by expression trees. After reading, you will be ready to use these structures to create rich adaptive algorithms.

1. [Expression Trees Explained](#)

Understand the structure and concepts behind *Expression Trees*.

2. [Framework Types Supporting Expression Trees](#)

Learn about the structures and classes that define and manipulate expression trees.

3. [Executing Expressions](#)

Learn how to convert an expression tree represented as a Lambda Expression into a delegate and execute the resulting delegate.

4. [Interpreting Expressions](#)

Learn how to traverse and examine *expression trees* to understand what code the expression tree represents.

5. [Building Expressions](#)

Learn how to construct the nodes for an expression tree and build expression trees.

6. [Translating Expressions](#)

Learn how to build a modified copy of an expression tree, or translate an expression tree into a different format.

7. [Summing up](#)

Review the information on expression trees.

Expression Trees Explained

10/29/2019 • 4 minutes to read • [Edit Online](#)

[Previous](#) -- [Overview](#)

An Expression Tree is a data structure that defines code. They are based on the same structures that a compiler uses to analyze code and generate the compiled output. As you read through this tutorial, you will notice quite a bit of similarity between Expression Trees and the types used in the Roslyn APIs to build [Analyzers and CodeFixes](#). (Analyzers and CodeFixes are NuGet packages that perform static analysis on code and can suggest potential fixes for a developer.) The concepts are similar, and the end result is a data structure that allows examination of the source code in a meaningful way. However, Expression Trees are based on a totally different set of classes and APIs than the Roslyn APIs.

Let's look at a simple example. Here's a line of code:

```
var sum = 1 + 2;
```

If you were to analyze this as an expression tree, the tree contains several nodes. The outermost node is a variable declaration statement with assignment (`var sum = 1 + 2;`) That outermost node contains several child nodes: a variable declaration, an assignment operator, and an expression representing the right hand side of the equals sign. That expression is further subdivided into expressions that represent the addition operation, and left and right operands of the addition.

Let's drill down a bit more into the expressions that make up the right side of the equals sign. The expression is `1 + 2`. That's a binary expression. More specifically, it's a binary addition expression. A binary addition expression has two children, representing the left and right nodes of the addition expression. Here, both nodes are constant expressions: The left operand is the value `1`, and the right operand is the value `2`.

Visually, the entire statement is a tree: You could start at the root node, and travel to each node in the tree to see the code that makes up the statement:

- Variable declaration statement with assignment (`var sum = 1 + 2;`)
 - Implicit variable type declaration (`var`)
 - Implicit var keyword (`var`)
 - Variable name declaration (`sum`)
 - Assignment operator (`=`)
 - Binary addition expression (`1 + 2`)
 - Left operand (`1`)
 - Addition operator (`+`)
 - Right operand (`2`)

This may look complicated, but it is very powerful. Following the same process, you can decompose much more complicated expressions. Consider this expression:

```
var finalAnswer = this.SecretSauceFunction(  
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),  
    decisionServer.considerFinalOptions("hello") +  
    MoreSecretSauce('A', DateTime.Now, true);
```

The expression above is also a variable declaration with an assignment. In this instance, the right hand side of the assignment is a much more complicated tree. I'm not going to decompose this expression, but consider what the different nodes might be. There are method calls using the current object as a receiver, one that has an explicit `this` receiver, one that does not. There are method calls using other receiver objects, there are constant arguments of different types. And finally, there is a binary addition operator. Depending on the return type of `SecretSauceFunction()` or `MoreSecretSauce()`, that binary addition operator may be a method call to an overridden addition operator, resolving to a static method call to the binary addition operator defined for a class.

Despite this perceived complexity, the expression above creates a tree structure that can be navigated as easily as the first sample. You can keep traversing child nodes to find leaf nodes in the expression. Parent nodes will have references to their children, and each node has a property that describes what kind of node it is.

The structure of an expression tree is very consistent. Once you've learned the basics, you can understand even the most complex code when it is represented as an expression tree. The elegance in the data structure explains how the C# compiler can analyze the most complex C# programs and create proper output from that complicated source code.

Once you become familiar with the structure of expression trees, you will find that knowledge you've gained quickly enables you to work with many more and more advanced scenarios. There is incredible power to expression trees.

In addition to translating algorithms to execute in other environments, expression trees can be used to make it easier to write algorithms that inspect code before executing it. You can write a method whose arguments are expressions and then examine those expressions before executing the code. The Expression Tree is a full representation of the code: you can see values of any sub-expression. You can see method and property names. You can see the value of any constant expressions. You can also convert an expression tree into an executable delegate, and execute the code.

The APIs for Expression Trees enable you to create trees that represent almost any valid code construct. However, to keep things as simple as possible, some C# idioms cannot be created in an expression tree. One example is asynchronous expressions (using the `async` and `await` keywords). If your needs require asynchronous algorithms, you would need to manipulate the `Task` objects directly, rather than rely on the compiler support. Another is in creating loops. Typically, you create these by using `for`, `foreach`, `while` or `do` loops. As you'll see [later in this series](#), the APIs for expression trees support a single loop expression, with `break` and `continue` expressions that control repeating the loop.

The one thing you can't do is modify an expression tree. Expression Trees are immutable data structures. If you want to mutate (change) an expression tree, you must create a new tree that is a copy of the original, but with your desired changes.

[Next -- Framework Types Supporting Expression Trees](#)

Framework Types Supporting Expression Trees

11/2/2020 • 3 minutes to read • [Edit Online](#)

[Previous -- Expression Trees Explained](#)

There is a large list of classes in the .NET Core framework that work with Expression Trees. You can see the full list at [System.Linq.Expressions](#). Rather than run through the full list, let's understand how the framework classes have been designed.

In language design, an expression is a body of code that evaluates and returns a value. Expressions may be very simple: the constant expression `1` returns the constant value of 1. They may be more complicated: The expression `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` returns one root for a quadratic equation (in the case where the equation has a solution).

It all starts with System.Linq.Expression

One of the complexities of working with expression trees is that many different kinds of expressions are valid in many places in programs. Consider an assignment expression. The right hand side of an assignment could be a constant value, a variable, a method call expression, or others. That language flexibility means that you may encounter many different expression types anywhere in the nodes of a tree when you traverse an expression tree. Therefore, when you can work with the base expression type, that's the simplest way to work. However, sometimes you need to know more. The base Expression class contains a `NodeType` property for this purpose. It returns an `ExpressionType` which is an enumeration of possible expression types. Once you know the type of the node, you can cast it to that type, and perform specific actions knowing the type of the expression node. You can search for certain node types, and then work with the specific properties of that kind of expression.

For example, this code will print the name of a variable for a variable access expression. I've followed the practice of checking the node type, then casting to a variable access expression and then checking the properties of the specific expression type:

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Creating Expression Trees

The `System.Linq.Expression` class also contains many static methods to create expressions. These methods create an expression node using the arguments supplied for its children. In this way, you build an expression up from its leaf nodes. For example, this code builds an Add expression:

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

You can see from this simple example that many types are involved in creating and working with expression trees. That complexity is necessary to provide the capabilities of the rich vocabulary provided by the C# language.

Navigating the APIs

There are Expression node types that map to almost all of the syntax elements of the C# language. Each type has specific methods for that type of language element. It's a lot to keep in your head at one time. Rather than try to memorize everything, here are the techniques I use to work with Expression trees:

1. Look at the members of the `ExpressionType` enum to determine possible nodes you should be examining.
This really helps when you want to traverse and understand an expression tree.
2. Look at the static members of the `Expression` class to build an expression. Those methods can build any expression type from a set of its child nodes.
3. Look at the `ExpressionVisitor` class to build a modified expression tree.

You'll find more as you look at each of those three areas. Invariably, you will find what you need when you start with one of those three steps.

[Next -- Executing Expression Trees](#)

Executing Expression Trees

3/12/2020 • 6 minutes to read • [Edit Online](#)

[Previous -- Framework Types Supporting Expression Trees](#)

An *expression tree* is a data structure that represents some code. It is not compiled and executable code. If you want to execute the .NET code that is represented by an expression tree, you must convert it into executable IL instructions.

Lambda Expressions to Functions

You can convert any `LambdaExpression`, or any type derived from `LambdaExpression` into executable IL. Other expression types cannot be directly converted into code. This restriction has little effect in practice. Lambda expressions are the only types of expressions that you would want to execute by converting to executable intermediate language (IL). (Think about what it would mean to directly execute a `ConstantExpression`. Would it mean anything useful?) Any expression tree that is a `LambdaExpression`, or a type derived from `LambdaExpression` can be converted to IL. The expression type `Expression<TDelegate>` is the only concrete example in the .NET Core libraries. It's used to represent an expression that maps to any delegate type. Because this type maps to a delegate type, .NET can examine the expression, and generate IL for an appropriate delegate that matches the signature of the lambda expression.

In most cases, this creates a simple mapping between an expression, and its corresponding delegate. For example, an expression tree that is represented by `Expression<Func<int>>` would be converted to a delegate of the type `Func<int>`. For a lambda expression with any return type and argument list, there exists a delegate type that is the target type for the executable code represented by that lambda expression.

The `LambdaExpression` type contains `Compile` and `CompileToMethod` members that you would use to convert an expression tree to executable code. The `Compile` method creates a delegate. The `CompileToMethod` method updates a `MethodBuilder` object with the IL that represents the compiled output of the expression tree. Note that `CompileToMethod` is only available in the full desktop framework, not in the .NET Core.

Optionally, you can also provide a `DebugInfoGenerator` that will receive the symbol debugging information for the generated delegate object. This enables you to convert the expression tree into a delegate object, and have full debugging information about the generated delegate.

You would convert an expression into a delegate using the following code:

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Notice that the delegate type is based on the expression type. You must know the return type and the argument list if you want to use the delegate object in a strongly typed manner. The `LambdaExpression.Compile()` method returns the `Delegate` type. You will have to cast it to the correct delegate type to have any compile-time tools check the argument list or return type.

Execution and Lifetimes

You execute the code by invoking the delegate created when you called `LambdaExpression.Compile()`. You can see this above where `add.Compile()` returns a delegate. Invoking that delegate, by calling `func()` executes the code.

That delegate represents the code in the expression tree. You can retain the handle to that delegate and invoke it later. You don't need to compile the expression tree each time you want to execute the code it represents. (Remember that expression trees are immutable, and compiling the same expression tree later will create a delegate that executes the same code.)

I will caution you against trying to create any more sophisticated caching mechanisms to increase performance by avoiding unnecessary compile calls. Comparing two arbitrary expression trees to determine if they represent the same algorithm will also be time consuming to execute. You'll likely find that the compute time you save avoiding any extra calls to `LambdaExpression.Compile()` will be more than consumed by the time executing code that determines of two different expression trees result in the same executable code.

Caveats

Compiling a lambda expression to a delegate and invoking that delegate is one of the simplest operations you can perform with an expression tree. However, even with this simple operation, there are caveats you must be aware of.

Lambda Expressions create closures over any local variables that are referenced in the expression. You must guarantee that any variables that would be part of the delegate are usable at the location where you call `Compile`, and when you execute the resulting delegate.

In general, the compiler will ensure that this is true. However, if your expression accesses a variable that implements `IDisposable`, it's possible that your code might dispose of the object while it is still held by the expression tree.

For example, this code works fine, because `int` does not implement `IDisposable`:

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

The delegate has captured a reference to the local variable `constant`. That variable is accessed at any time later, when the function returned by `CreateBoundFunc` executes.

However, consider this (rather contrived) class that implements `IDisposable`:

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

If you use it in an expression as shown below, you'll get an `ObjectDisposedException` when you execute the code referenced by the `Resource.Argument` property:

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

The delegate returned from this method has closed over the `constant` object, which has been disposed of. (It's been disposed, because it was declared in a `using` statement.)

Now, when you execute the delegate returned from this method, you'll have a `ObjectDisposedException` thrown at the point of execution.

It does seem strange to have a runtime error representing a compile-time construct, but that's the world we enter when we work with expression trees.

There are a lot of permutations of this problem, so it's hard to offer general guidance to avoid it. Be careful about accessing local variables when defining expressions, and be careful about accessing state in the current object (represented by `this`) when creating an expression tree that can be returned by a public API.

The code in your expression may reference methods or properties in other assemblies. That assembly must be accessible when the expression is defined, and when it is compiled, and when the resulting delegate is invoked. You'll be met with a `ReferencedAssemblyNotFoundException` in cases where it is not present.

Summary

Expression Trees that represent lambda expressions can be compiled to create a delegate that you can execute. This provides one mechanism to execute the code represented by an expression tree.

The Expression Tree does represent the code that would execute for any given construct you create. As long as the environment where you compile and execute the code matches the environment where you create the expression, everything works as expected. When that doesn't happen, the errors are very predictable, and they will be caught in your first tests of any code using the expression trees.

[Next -- Interpreting Expressions](#)

Interpreting Expressions

11/2/2020 • 14 minutes to read • [Edit Online](#)

[Previous -- Executing Expressions](#)

Now, let's write some code to examine the structure of an *expression tree*. Every node in an expression tree will be an object of a class that is derived from `Expression`.

That design makes visiting all the nodes in an expression tree a relatively straight forward recursive operation. The general strategy is to start at the root node and determine what kind of node it is.

If the node type has children, recursively visit the children. At each child node, repeat the process used at the root node: determine the type, and if the type has children, visit each of the children.

Examining an Expression with No Children

Let's start by visiting each node in a simple expression tree. Here's the code that creates a constant expression and then examines its properties:

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

This will print the following:

```
This is an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Now, let's write the code that would examine this expression and write out some important properties about it. Here's that code:

Examining a simple Addition Expression

Let's start with the addition sample from the introduction to this section.

```
Expression<Func<int>> sum = () => 1 + 2;
```

I'm not using `var` to declare this expression tree, as it is not possible because the right-hand side of the assignment is implicitly typed.

The root node is a `LambdaExpression`. In order to get the interesting code on the right-hand side of the `=>` operator, you need to find one of the children of the `LambdaExpression`. We'll do that with all the expressions in this section. The parent node does help us find the return type of the `LambdaExpression`.

To examine each node in this expression, we'll need to recursively visit a number of nodes. Here's a simple first implementation:

```

Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They are:");
foreach(var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"{Environment.NewLine}\tParameter Type: {argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"{Environment.NewLine}\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"{Environment.NewLine}\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");

```

This sample prints the following output:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
    Parameter Type: System.Int32, Name: a
    Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
    Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
    Parameter Type: System.Int32, Name: b

```

You'll notice a lot of repetition in the code sample above. Let's clean that up and build a more general purpose expression node visitor. That's going to require us to write a recursive algorithm. Any node could be of a type that might have children. Any node that has children requires us to visit those children and determine what that node is. Here's the cleaned up version that utilizes recursion to visit the addition operations:

```

// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node)
    {
        this.node = node;
    }

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
                return new ParameterVisitor((ParameterExpression)node);
        }
    }
}

```

```

        case ExpressionType.Add:
            return new BinaryVisitor((BinaryExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name == null) ? "<null>" :
node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)

```

```

    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, Name: {node.Name}, ByRef:
{node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is {node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is {node.Value}");
    }
}

```

This algorithm is the basis of an algorithm that can visit any arbitrary `LambdaExpression`. There are many holes, namely that the code I created only looks for a very small sample of the possible sets of expression tree nodes that it may encounter. However, you can still learn quite a bit from what it produces. (The default case in the `Visitor.CreateFromExpression` method prints a message to the error console when a new node type is encountered. That way, you know to add a new expression type.)

When you run this visitor on the addition expression shown above, you get the following output:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False

```

Now that you've built a more general visitor implementation, you can visit and process many more different types of expressions.

Examining an Addition Expression with Many Levels

Let's try a more complicated example, yet still limit the node types to addition only:

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Before you run this on the visitor algorithm, try a thought exercise to work out what the output might be. Remember that the `+` operator is a *binary operator*: it must have two children, representing the left and right operands. There are several possible ways to construct a tree that could be correct:

```

Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;

```

You can see the separation into two possible answers to highlight the most promising. The first represents *right associative* expressions. The second represent *left associative* expressions. The advantage of both of those two formats is that the format scales to any arbitrary number of addition expressions.

If you do run this expression through the visitor, you will see this output, verifying that the simple addition expression is *left associative*.

In order to run this sample, and see the full expression tree, I had to make one change to the source expression tree. When the expression tree contains all constants, the resulting tree simply contains the constant value of [10](#). The compiler performs all the addition and reduces the expression to its simplest form. Simply adding one variable in the expression is sufficient to see the original tree:

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Create a visitor for this sum and run the visitor you'll see this output:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
The expression body is:
    This binary expression is a Add expression
        The Left argument is:
            This binary expression is a Add expression
                The Left argument is:
                    This binary expression is a Add expression
                        The Left argument is:
                            This is an Constant expression type
                            The type of the constant value is System.Int32
                            The value of the constant value is 1
                    The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                    This is an Constant expression type
                    The type of the constant value is System.Int32
                    The value of the constant value is 3
        The Right argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 4

```

You can also run any of the other samples through the visitor code and see what tree it represents. Here's an example of the [sum3](#) expression above (with an additional parameter to prevent the compiler from computing the constant):

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Here's the output from the visitor:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This binary expression is a Add expression
        The Left argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 1
        The Right argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
        This binary expression is a Add expression
        The Left argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 3
        The Right argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: b, ByRef: False

```

Notice that the parentheses are not part of the output. There are no nodes in the expression tree that represent the parentheses in the input expression. The structure of the expression tree contains all the information necessary to communicate the precedence.

Extending from this sample

The sample deals with only the most rudimentary expression trees. The code you've seen in this section only handles constant integers and the binary `+</code>` operator. As a final sample, let's update the visitor to handle a more complicated expression. Let's make it work for this:

```

Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);

```

This code represents one possible implementation for the mathematical *factorial* function. The way I've written this code highlights two limitations of building expression trees by assigning lambda expressions to Expressions. First, statement lambdas are not allowed. That means I can't use loops, blocks, if / else statements, and other control structures common in C#. I'm limited to using expressions. Second, I can't recursively call the same expression. I could if it were already a delegate, but I can't call it in its expression tree form. In the section on [building expression trees](#), you'll learn techniques to overcome these limitations.

In this expression, you'll encounter nodes of all these types:

1. Equal (binary expression)
2. Multiply (binary expression)
3. Conditional (the `? : expression`)
4. Method Call Expression (calling `Range()` and `Aggregate()`)

One way to modify the visitor algorithm is to keep executing it, and write the node type every time you reach your `default` clause. After a few iterations, you'll have seen each of the potential nodes. Then, you have all you need. The result would be something like this:

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

The `ConditionalVisitor` and `MethodCallVisitor` process those two nodes:

```

public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach(var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}

```

And the output for the expression tree would be:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: n, ByRef: False
The expression body is:
    This expression is a Conditional expression
    The Test for this expression is:
        This binary expression is a Equal expression
        The Left argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: n, ByRef: False
        The Right argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 0
    The True clause for this expression is:
        This is an Constant expression type
        The type of the constant value is System.Int32
        The value of the constant value is 1
    The False clause for this expression is:
        This expression is a Call expression
        This is a static method call
        The method name is System.Linq.Enumerable.Aggregate
        The Arguments are:
            This expression is a Call expression
            This is a static method call
            The method name is System.Linq.Enumerable.Range
        The Arguments are:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 1
            This is an Parameter expression type
            Type: System.Int32, Name: n, ByRef: False
    This expression is a Lambda expression type
    The name of the lambda is <null>
    The return type is System.Int32
    The expression has 2 arguments. They are:
        This is an Parameter expression type
        Type: System.Int32, Name: product, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: factor, ByRef: False
    The expression body is:
        This binary expression is a Multiply expression
        The Left argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: product, ByRef: False
        The Right argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: factor, ByRef: False

```

Extending the Sample Library

The samples in this section show the core techniques to visit and examine nodes in an expression tree. I glossed over many actions you might need in order to concentrate on the core tasks of visiting and accessing nodes in an expression tree.

First, the visitors only handle constants that are integers. Constant values could be any other numeric type, and the C# language supports conversions and promotions between those types. A more robust version of this code would mirror all those capabilities.

Even the last example recognizes a subset of the possible node types. You can still feed it many expressions that will cause it to fail. A full implementation is included in .NET Standard under the name [ExpressionVisitor](#) and can

handle all the possible node types.

Finally, the library I used in this article was built for demonstration and learning. It's not optimized. I wrote it to make the structures used clear, and to highlight the techniques used to visit the nodes and analyze what's there. A production implementation would pay more attention to performance than I have.

Even with those limitations, you should be well on your way to writing algorithms that read and understand expression trees.

[Next -- Building Expressions](#)

Building Expression Trees

3/23/2021 • 5 minutes to read • [Edit Online](#)

[Previous -- Interpreting Expressions](#)

All the expression trees you've seen so far have been created by the C# compiler. All you had to do was create a lambda expression that was assigned to a variable typed as an `Expression<Func<T>>` or some similar type. That's not the only way to create an expression tree. For many scenarios you may find that you need to build an expression in memory at runtime.

Building Expression Trees is complicated by the fact that those expression trees are immutable. Being immutable means that you must build the tree from the leaves up to the root. The APIs you'll use to build expression trees reflect this fact: The methods you'll use to build a node take all its children as arguments. Let's walk through a few examples to show you the techniques.

Creating Nodes

Let's start relatively simply again. We'll use the addition expression I've been working with throughout these sections:

```
Expression<Func<int>> sum = () => 1 + 2;
```

To construct that expression tree, you must construct the leaf nodes. The leaf nodes are constants, so you can use the `Expression.Constant` method to create the nodes:

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Next, you'll build the addition expression:

```
var addition = Expression.Add(one, two);
```

Once you've got the addition expression, you can create the lambda expression:

```
var lambda = Expression.Lambda(addition);
```

This is a very simple lambda expression, because it contains no arguments. Later in this section, you'll see how to map arguments to parameters and build more complicated expressions.

For expressions that are as simple as this one, you may combine all the calls into a single statement:

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Building a Tree

That's the basics of building an expression tree in memory. More complex trees generally mean more node types, and more nodes in the tree. Let's run through one more example and show two more node types that you will typically build when you create expression trees: the argument nodes, and method call nodes.

Let's build an expression tree to create this expression:

```
Expression<Func<double, double, double>> distanceCalc =  
(x, y) => Math.Sqrt(x * x + y * y);
```

You'll start by creating parameter expressions for `x` and `y`:

```
var xParameter = Expression.Parameter(typeof(double), "x");  
var yParameter = Expression.Parameter(typeof(double), "y");
```

Creating the multiplication and addition expressions follows the pattern you've already seen:

```
var xSquared = Expression.Multiply(xParameter, xParameter);  
var ySquared = Expression.Multiply(yParameter, yParameter);  
var sum = Expression.Add(xSquared, ySquared);
```

Next, you need to create a method call expression for the call to `Math.Sqrt`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });  
var distance = Expression.Call(sqrtMethod, sum);
```

And then finally, you put the method call into a lambda expression, and make sure to define the arguments to the lambda expression:

```
var distanceLambda = Expression.Lambda(  
    distance,  
    xParameter,  
    yParameter);
```

In this more complicated example, you see a couple more techniques that you will often need to create expression trees.

First, you need to create the objects that represent parameters or local variables before you use them. Once you've created those objects, you can use them in your expression tree wherever you need.

Second, you need to use a subset of the Reflection APIs to create a `MethodInfo` object so that you can create an expression tree to access that method. You must limit yourself to the subset of the Reflection APIs that are available on the .NET Core platform. Again, these techniques will extend to other expression trees.

Building Code In Depth

You aren't limited in what you can build using these APIs. However, the more complicated expression tree that you want to build, the more difficult the code is to manage and to read.

Let's build an expression tree that is the equivalent of this code:

```

Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};

```

Notice above that I did not build the expression tree, but simply the delegate. Using the `Expression` class, you can't build statement lambdas. Here's the code that is required to build the same functionality. It's complicated by the fact that there isn't an API to build a `while` loop, instead you need to build a loop that contains a conditional test, and a label target to break out of the loop.

```

var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);

```

The code to build the expression tree for the factorial function is quite a bit longer, more complicated, and it's riddled with labels and break statements and other elements we'd like to avoid in our everyday coding tasks.

For this section, I've also updated the visitor code to visit every node in this expression tree and write out information about the nodes that are created in this sample. You can [view or download the sample code](#) at the dotnet/docs GitHub repository. Experiment for yourself by building and running the samples. For download instructions, see [Samples and Tutorials](#).

Examining the APIs

The expression tree APIs are some of the more difficult to navigate in .NET Core, but that's fine. Their purpose is a rather complex undertaking: writing code that generates code at runtime. They are necessarily complicated to provide a balance between supporting all the control structures available in the C# language and keeping the surface area of the APIs as small as reasonable. This balance means that many control structures are

represented not by their C# constructs, but by constructs that represent the underlying logic that the compiler generates from these higher level constructs.

Also, at this time, there are C# expressions that cannot be built directly using `Expression` class methods. In general, these will be the newest operators and expressions added in C# 5 and C# 6. (For example, `async` expressions cannot be built, and the new `?.` operator cannot be directly created.)

[Next -- Translating Expressions](#)

Translate expression trees

11/2/2020 • 6 minutes to read • [Edit Online](#)

[Previous -- Building Expressions](#)

In this final section, you'll learn how to visit each node in an expression tree while building a modified copy of that expression tree. These are the techniques that you will use in two important scenarios. The first is to understand the algorithms expressed by an expression tree so that it can be translated into another environment. The second is when you want to change the algorithm that has been created. This might be to add logging, intercept method calls and track them, or other purposes.

Translating is Visiting

The code you build to translate an expression tree is an extension of what you've already seen to visit all the nodes in a tree. When you translate an expression tree, you visit all the nodes, and while visiting them, build the new tree. The new tree may contain references to the original nodes, or new nodes that you have placed in the tree.

Let's see this in action by visiting an expression tree, and creating a new tree with some replacement nodes. In this example, let's replace any constant with a constant that is ten times larger. Otherwise, we'll leave the expression tree intact. Rather than reading the value of the constant, and replacing it with a new constant, we'll make this replacement by replacing the constant node with a new node that performs the multiplication.

Here, once you find a constant node, you create a new multiplication node whose children are the original constant, and the constant `10`:

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

By replacing the original node with the substitute, a new tree is formed that contains our modifications. We can verify that by compiling and executing the replaced tree.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Building a new tree is a combination of visiting the nodes in the existing tree, and creating new nodes and inserting them into the tree.

This example shows the importance of expression trees being immutable. Notice that the new tree created above contains a mixture of newly created nodes, and nodes from the existing tree. That's safe, because the nodes in the existing tree cannot be modified. This can result in significant memory efficiencies. The same nodes can be used throughout a tree, or in multiple expression trees. Since nodes can't be modified, the same node can be reused whenever it's needed.

Traversing and Executing an Addition

Let's verify this by building a second visitor that walks the tree of addition nodes and computes the result. You can do this by making a couple modifications to the visitor that you've seen so far. In this new version, the visitor will return the partial sum of the addition operation up to this point. For a constant expression, that is simply the value of the constant expression. For an addition expression, the result is the sum of the left and right operands, once those trees have been traversed.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so we can call it
// from itself recursively:
Func<Expression, int> aggregate = null;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
        (int)((ConstantExpression)exp).Value :
        aggregate(((BinaryExpression)exp).Left) + aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

There's quite a bit of code here, but the concepts are very approachable. This code visits children in a depth first search. When it encounters a constant node, the visitor returns the value of the constant. After the visitor has visited both children, those children will have computed the sum computed for that subtree. The addition node can now compute its sum. Once all the nodes in the expression tree have been visited, the sum will have been computed. You can trace the execution by running the sample in the debugger and tracing the execution.

Let's make it easier to trace how the nodes are analyzed and how the sum is computed by traversing the tree. Here's an updated version of the Aggregate method that includes quite a bit of tracing information:

```

private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}

```

Running it on the same expression yields the following output:

```

10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10

```

Trace the output and follow along in the code above. You should be able to work out how the code visits each node and computes the sum as it goes through the tree and finds the sum.

Now, let's look at a different run, with the expression given by `sum1`:

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Here's the output from examining this expression:

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

While the final answer is the same, the tree traversal is completely different. The nodes are traveled in a different order, because the tree was constructed with different operations occurring first.

Learning More

This sample shows a small subset of the code you would build to traverse and interpret the algorithms represented by an expression tree. For a complete discussion of all the work necessary to build a general purpose library that translates expression trees into another language, please read [this series](#) by Matt Warren. It goes into great detail on how to translate any of the code you might find in an expression tree.

I hope you've now seen the true power of expression trees. You can examine a set of code, make any changes you'd like to that code, and execute the changed version. Because the expression trees are immutable, you can create new trees by using the components of existing trees. This minimizes the amount of memory needed to create modified expression trees.

[Next -- Summing up](#)

Expression Trees Summary

3/12/2020 • 2 minutes to read • [Edit Online](#)

[Previous -- Translating Expressions](#)

In this series, you've seen how you can use *expression trees* to create dynamic programs that interpret code as data and build new functionality based on that code.

You can examine expression trees to understand the intent of an algorithm. You can not only examine that code. You can build new expression trees that represent modified versions of the original code.

You can also use expression trees to look at an algorithm, and translate that algorithm into another language or environment.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees cannot contain `await` expressions, or `async` lambda expressions. Many of the features added in the C# 6 release don't appear exactly as written in expression trees. Instead, newer features will be exposed in expression trees in the equivalent, earlier syntax. This may not be as much of a limitation as you might think. In fact, it means that your code that interprets expression trees will likely still work the same when new language features are introduced.

Even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It's a powerful tool, and it's one of the features of the .NET ecosystem that enables rich libraries such as Entity Framework to accomplish what they do.

Interoperability (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

[Interoperability Overview](#)

Describes methods to interoperate between C# managed code and unmanaged code.

[How to access Office interop objects by using C# features](#)

Describes features that are introduced in Visual C# to facilitate Office programming.

[How to use indexed properties in COM interop programming](#)

Describes how to use indexed properties to access COM properties that have parameters.

[How to use platform invoke to play a WAV file](#)

Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

[Walkthrough: Office Programming](#)

Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

[Example COM Class](#)

Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see [Basic concepts](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Marshal.ReleaseComObject](#)
- [C# Programming Guide](#)
- [Interoperating with Unmanaged Code](#)
- [Walkthrough: Office Programming](#)

Document your C# code with XML comments

3/15/2021 • 28 minutes to read • [Edit Online](#)

XML documentation comments are a special kind of comment, added above the definition of any user-defined type or member. They are special because they can be processed by the compiler to generate an XML documentation file at compile time. The compiler-generated XML file can be distributed alongside your .NET assembly so that Visual Studio and other IDEs can use IntelliSense to show quick information about types or members. Additionally, the XML file can be run through tools like [DocFX](#) and [Sandcastle](#) to generate API reference websites.

XML documentation comments, like all other comments, are ignored by the compiler.

You can generate the XML file at compile time by doing one of the following:

- If you are developing an application with .NET Core from the command line, you can add a `GenerateDocumentationFile` element to the `<PropertyGroup>` section of your .csproj project file. You can also specify the path to the documentation file directly using `DocumentationFile` element. The following example generates an XML file in the project directory with the same root filename as the assembly:

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

This is equivalent to the following:

```
<DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
```

- If you are developing an application using Visual Studio, right-click on the project and select **Properties**. In the properties dialog, select the **Build** tab, and check **XML documentation file**. You can also change the location to which the compiler writes the file.
- If you are compiling a .NET application from the command line, add the [DocumentationFile compiler option](#) when compiling.

XML documentation comments use triple forward slashes (`///`) and an XML formatted comment body. For example:

```
/// <summary>
/// This class does something.
/// </summary>
public class SomeClass
{}
```

Walkthrough

Let's walk through documenting a very basic math library to make it easy for new developers to understand/contribute and for third-party developers to use.

Here's code for the simple math library:

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
public class Math
{
    // Adds two integers and returns the result
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

The sample library supports four major arithmetic operations (`add`, `subtract`, `multiply`, and `divide`) on `int` and `double` data types.

Now you want to be able to create an API reference document from your code for third-party developers who use your library but don't have access to the source code. As mentioned earlier XML documentation tags can be used to achieve this. You will now be introduced to the standard XML tags the C# compiler supports.

<summary>

The `<summary>` tag adds brief information about a type or member. I'll demonstrate its use by adding it to the `Math` class definition and the first `Add` method. Feel free to apply it to the rest of your code.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

The `<summary>` tag is important, and we recommend that you include it because its content is the primary source of type or member information in IntelliSense or an API reference document.

<remarks>

The `<remarks>` tag supplements the information about types or members that the `<summary>` tag provides. In this example, you'll just add it to the class.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// </remarks>
public class Math
{
```

<returns>

The `<returns>` tag describes the return value of a method declaration. As before, the following example

illustrates the `<returns>` tag on the first `Add` method. You can do the same on other methods.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

<value>

The `<value>` tag is similar to the `<returns>` tag, except that you use it for properties. Assuming your `Math` library had a static property called `PI`, here's how you'd use this tag:

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// These operations can be performed on both integers and doubles
/// </remarks>
public class Math
{
    /// <value>Gets the value of PI.</value>
    public static double PI { get; }
}
```

<example>

You use the `<example>` tag to include an example in your XML documentation. This involves using the child `<code>` tag.

```

// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Add(4, 5);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}

```

The `code` tag preserves line breaks and indentation for longer examples.

<para>

You use the `<para>` tag to format the content within its parent tag. `<para>` is usually used inside a tag, such as `<remarks>` or `<returns>`, to divide text into paragraphs. You can format the contents of the `<remarks>` tag for your class definition.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
}

```

<C>

Still on the topic of formatting, you use the `<c>` tag for marking part of text as code. It's like the `<code>` tag but inline. It's useful when you want to show a quick code example as part of a tag's content. Let's update the documentation for the `Math` class.

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
}

```

<exception>

By using the `<exception>` tag, you let your developers know that a method can throw specific exceptions.

Looking at your `Math` library, you can see that both `Add` methods throw an exception if a certain condition is met. Not so obvious, though, is that integer `Divide` method throws as well if the `b` parameter is zero. Now add exception documentation to this method.

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    public static double Add(double a, double b)
    {

```

```

        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Divides an integer by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two integers.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    /// <summary>
    /// Divides a double by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two doubles.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

The `cref` attribute represents a reference to an exception that is available from the current compilation environment. This can be any type defined in the project or a referenced assembly. The compiler will issue a warning if its value cannot be resolved.

<see>

The `<see>` tag lets you create a clickable link to a documentation page for another code element. In our next example, we'll create a clickable link between the two `Add` methods.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

The `ref` is a **required** attribute that represents a reference to a type or its member that is available from the current compilation environment. This can be any type defined in the project or a referenced assembly.

<seealso>

You use the `<seealso>` tag in the same way you do the `<see>` tag. The only difference is that its content is typically placed in a "See Also" section. Here we'll add a `seealso` tag on the integer `Add` method to reference other methods in the class that accept integer parameters:

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <seealso cref="Math.Divide(int, int)">
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

The `cref` attribute represents a reference to a type or its member that is available from the current compilation environment. This can be any type defined in the project or a referenced assembly.

<param>

You use the `<param>` tag to describe a method's parameters. Here's an example on the `double Add` method: The parameter the tag describes is specified in the required `name` attribute.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

<typeparam>

You use `<typeparam>` tag just like the `<param>` tag but for generic type or method declarations to describe a generic parameter. Add a quick generic method to your `Math` class to check if one quantity is greater than another.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}

```

<paramref>

Sometimes you might be in the middle of describing what a method does in what could be a `<summary>` tag, and you might want to make a reference to a parameter. The `<paramref>` tag is great for just this. Let's update the summary of our double based `Add` method. Like the `<param>` tag, the parameter name is specified in the

required `name` attribute.

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparamref>

You use `<typeparamref>` tag just like the `<paramref>` tag but for generic type or method declarations to describe a generic parameter. You can use the same generic method you previously created.

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable <typeparamref name="T"/> is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<list>

You use the `<list>` tag to format documentation information as an ordered list, unordered list, or table. Make an unordered list of every math operation your `Math` library supports.

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
public class Math
{
}

```

You can make an ordered list or table by changing the `type` attribute to `number` or `table`, respectively.

<inheritdoc>

You can use the `<inheritdoc>` tag to inherit XML comments from base classes, interfaces, and similar methods. This eliminates unwanted copying and pasting of duplicate XML comments and automatically keeps XML comments synchronized.

```

/*
   The IMath interface
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// This is the IMath interface.
/// </summary>
public interface IMath
{
}

/// <inheritdoc/>
public class Math : IMath
{
}

```

Put it all together

If you've followed this tutorial and applied the tags to your code where necessary, your code should now look similar to the following:

```

/*
   The main Math class
   Contains all methods for performing basic math functions

```

```

    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <seealso cref="Math.Divide(int, int)">
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// </returns>
}

```

```
/// The sum of two doubles.  
/// </returns>  
/// <example>  
/// <code>  
/// double c = Math.Add(4.5, 5.4);  
/// if (c > 10)  
/// {  
///     Console.WriteLine(c);  
/// }  
/// </code>  
/// </example>  
/// <exception cref="System.OverflowException">Thrown when one parameter is max  
/// and the other is greater than 0.</exception>  
/// See <see cref="Math.Add(int, int)" /> to add integers.  
/// <seealso cref="Math.Subtract(double, double)" />  
/// <seealso cref="Math.Multiply(double, double)" />  
/// <seealso cref="Math.Divide(double, double)" />  
/// <param name="a">A double precision number.</param>  
/// <param name="b">A double precision number.</param>  
public static double Add(double a, double b)  
{  
    // If any parameter is equal to the max value of an integer  
    // and the other is greater than zero  
    if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))  
        throw new System.OverflowException();  
  
    return a + b;  
}  
  
// Subtracts an integer from another and returns the result  
/// <summary>  
/// Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.  
/// </summary>  
/// <returns>  
/// The difference between two integers.  
/// </returns>  
/// <example>  
/// <code>  
/// int c = Math.Subtract(4, 5);  
/// if (c > 1)  
/// {  
///     Console.WriteLine(c);  
/// }  
/// </code>  
/// </example>  
/// See <see cref="Math.Subtract(double, double)" /> to subtract doubles.  
/// <seealso cref="Math.Add(int, int)" />  
/// <seealso cref="Math.Multiply(int, int)" />  
/// <seealso cref="Math.Divide(int, int)" />  
/// <param name="a">An integer.</param>  
/// <param name="b">An integer.</param>  
public static int Subtract(int a, int b)  
{  
    return a - b;  
}  
  
// Subtracts a double from another and returns the result  
/// <summary>  
/// Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the  
result.  
/// </summary>  
/// <returns>  
/// The difference between two doubles.  
/// </returns>  
/// <example>  
/// <code>  
/// double c = Math.Subtract(4.5, 5.4);  
/// if (c > 1)  
/// {
```

```
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(int, int)"> to subtract integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Multiply(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Subtract(double a, double b)
{
    return a - b;
}

// Multiplies two integers and returns the result
/// <summary>
/// Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Multiply(4, 5);
/// if (c > 100)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(double, double)"> to multiply doubles.
/// <seealso cref="Math.Add(int, int)">
/// <seealso cref="Math.Subtract(int, int)">
/// <seealso cref="Math.Divide(int, int)">
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Multiply(int a, int b)
{
    return a * b;
}

// Multiplies two doubles and returns the result
/// <summary>
/// Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Multiply(4.5, 5.4);
/// if (c > 100.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(int, int)"> to multiply integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Subtract(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Multiply(double a, double b)
{
    return a * b;
}
```

```

// Divides an integer by another and returns the result
/// <summary>
/// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the
result.
/// </summary>
/// <returns>
/// The quotient of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Divide(4, 5);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
/// See <see cref="Math.Divide(double, double)"> to divide doubles.
/// <seealso cref="Math.Add(int, int)">
/// <seealso cref="Math.Subtract(int, int)">
/// <seealso cref="Math.Multiply(int, int)">
/// <param name="a">An integer dividend.</param>
/// <param name="b">An integer divisor.</param>
public static int Divide(int a, int b)
{
    return a / b;
}

// Divides a double by another and returns the result
/// <summary>
/// Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The quotient of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Divide(4.5, 5.4);
/// if (c > 1.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
/// See <see cref="Math.Divide(int, int)"> to divide integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Subtract(double, double)">
/// <seealso cref="Math.Multiply(double, double)">
/// <param name="a">A double precision dividend.</param>
/// <param name="b">A double precision divisor.</param>
public static double Divide(double a, double b)
{
    return a / b;
}

```

From your code, you can generate a detailed documentation website complete with clickable cross-references. But you're faced with another problem: your code has become hard to read. There's so much information to sift through that this is going to be a nightmare for any developer who wants to contribute to this code. Thankfully there's an XML tag that can help you deal with this:

<include>

The `<include>` tag lets you refer to comments in a separate XML file that describe the types and members in your source code, as opposed to placing documentation comments directly in your source code file.

Now you're going to move all your XML tags into a separate XML file named `docs.xml`. Feel free to name the file whatever you want.

```
<docs>
  <members name="math">
    <Math>
      <summary>
        The main <c>Math</c> class.
        Contains all methods for performing basic math functions.
      </summary>
      <remarks>
        <para>This class can add, subtract, multiply and divide.</para>
        <para>These operations can be performed on both integers and doubles.</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
        Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
      </summary>
      <returns>
        The sum of two integers.
      </returns>
      <example>
        <code>
          int c = Math.Add(4, 5);
          if (c > 10)
          {
            Console.WriteLine(c);
          }
        </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
        and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"> to add doubles.
      <seealso cref="Math.Subtract(int, int)">
      <seealso cref="Math.Multiply(int, int)">
      <seealso cref="Math.Divide(int, int)">
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
        Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
      </summary>
      <returns>
        The sum of two doubles.
      </returns>
      <example>
        <code>
          double c = Math.Add(4.5, 5.4);
          if (c > 10)
          {
            Console.WriteLine(c);
          }
        </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
        and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"> to add integers.
      <seealso cref="Math.Subtract(double, double)">
      <seealso cref="Math.Multiply(double, double)">
      <seealso cref="Math.Divide(double, double)">
```

```
<seealso cref="Math.Divide(double, double)" />
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</AddDouble>
<SubtractInt>
    <summary>
        Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
    </summary>
    <returns>
        The difference between two integers.
    </returns>
    <example>
        <code>
            int c = Math.Subtract(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
        </code>
    </example>
    See <see cref="Math.Subtract(double, double)" /> to subtract doubles.
    <seealso cref="Math.Add(int, int)" />
    <seealso cref="Math.Multiply(int, int)" />
    <seealso cref="Math.Divide(int, int)" />
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
</SubtractInt>
<SubtractDouble>
    <summary>
        Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
    </summary>
    <returns>
        The difference between two doubles.
    </returns>
    <example>
        <code>
            double c = Math.Subtract(4.5, 5.4);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
        </code>
    </example>
    See <see cref="Math.Subtract(int, int)" /> to subtract integers.
    <seealso cref="Math.Add(double, double)" />
    <seealso cref="Math.Multiply(double, double)" />
    <seealso cref="Math.Divide(double, double)" />
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
</SubtractDouble>
<MultiplyInt>
    <summary>
        Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    </summary>
    <returns>
        The product of two integers.
    </returns>
    <example>
        <code>
            int c = Math.Multiply(4, 5);
            if (c > 100)
            {
                Console.WriteLine(c);
            }
        </code>
    </example>
    See <see cref="Math.Multiply(double, double)" /> to multiply doubles.
    <seealso cref="Math.Add(int, int)" />
```

```
<seealso cref="Math.Subtract(int, int)" />
<seealso cref="Math.Divide(int, int)" />
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</MultiplyInt>
<MultiplyDouble>
    <summary>
        Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    </summary>
    <returns>
        The product of two doubles.
    </returns>
    <example>
        <code>
            double c = Math.Multiply(4.5, 5.4);
            if (c > 100.0)
            {
                Console.WriteLine(c);
            }
        </code>
    </example>
    See <see cref="Math.Multiply(int, int)" /> to multiply integers.
    <seealso cref="Math.Add(double, double)" />
    <seealso cref="Math.Subtract(double, double)" />
    <seealso cref="Math.Divide(double, double)" />
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
</MultiplyDouble>
<DivideInt>
    <summary>
        Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the result.
    </summary>
    <returns>
        The quotient of two integers.
    </returns>
    <example>
        <code>
            int c = Math.Divide(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
        </code>
    </example>
    <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    See <see cref="Math.Divide(double, double)" /> to divide doubles.
    <seealso cref="Math.Add(int, int)" />
    <seealso cref="Math.Subtract(int, int)" />
    <seealso cref="Math.Multiply(int, int)" />
    <param name="a">An integer dividend.</param>
    <param name="b">An integer divisor.</param>
</DivideInt>
<DivideDouble>
    <summary>
        Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
    </summary>
    <returns>
        The quotient of two doubles.
    </returns>
    <example>
        <code>
            double c = Math.Divide(4.5, 5.4);
            if (c > 1.0)
            {
                Console.WriteLine(c);
            }
        </code>
    </example>

```

```

</code>
</example>
<exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    See <see cref="Math.Divide(int, int)"> to divide integers.
    <seealso cref="Math.Add(double, double)">
    <seealso cref="Math.Subtract(double, double)">
    <seealso cref="Math.Multiply(double, double)">
    <param name="a">A double precision dividend.</param>
    <param name="b">A double precision divisor.</param>
</DivideDouble>
</members>
</docs>

```

In the above XML, each member's documentation comments appear directly inside a tag named after what they do. You can choose your own strategy. Now that you have your XML comments in a separate file, let's see how your code can be made more readable by using the `<include>` tag:

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <include file='docs.xml' path='docs/members[@name="math"]/Math/*'>
public class Math
{
    // Adds two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddDouble/*'>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractInt/*'>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractDouble/*'>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyInt/*'>
    public static int Multiply(int a, int b)
    {

```

```

        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyDouble/*'>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideInt/*'>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideDouble/*'>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

And there you have it: our code is back to being readable, and no documentation information has been lost.

The `file` attribute represents the name of the XML file containing the documentation.

The `path` attribute represents an [XPath](#) query to the `tag name` present in the specified `file`.

The `name` attribute represents the name specifier in the tag that precedes the comments.

The `id` attribute, which can be used in place of `name`, represents the ID for the tag that precedes the comments.

User-defined tags

All the tags outlined above represent those that are recognized by the C# compiler. However, a user is free to define their own tags. Tools like Sandcastle bring support for extra tags like `<event>` and `<note>`, and even support [documenting namespaces](#). Custom or in-house documentation generation tools can also be used with the standard tags and multiple output formats from HTML to PDF can be supported.

Recommendations

Documenting code is recommended for many reasons. What follows are some best practices, general use case scenarios, and things that you should know when using XML documentation tags in your C# code.

- For the sake of consistency, all publicly visible types and their members should be documented. If you must do it, do it all.
- Private members can also be documented using XML comments. However, this exposes the inner (potentially confidential) workings of your library.
- At a bare minimum, types and their members should have a `<summary>` tag because its content is needed for IntelliSense.
- Documentation text should be written using complete sentences ending with full stops.
- Partial classes are fully supported, and documentation information will be concatenated into a single entry for that type.
- The compiler verifies the syntax of the `<exception>`, `<include>`, `<param>`, `<see>`, `<seealso>`, and `<typeparam>` tags.
- The compiler validates the parameters that contain file paths and references to other parts of the code.

See also

- [XML Documentation Comments \(C# Programming Guide\)](#)
- [Recommended Tags for Documentation Comments \(C# Programming Guide\)](#)

Versioning in C#

4/20/2020 • 5 minutes to read • [Edit Online](#)

In this tutorial you'll learn what versioning means in .NET. You'll also learn the factors to consider when versioning your library as well as upgrading to a new version of a library.

Authoring Libraries

As a developer who has created .NET libraries for public use, you've most likely been in situations where you have to roll out new updates. How you go about this process matters a lot as you need to ensure that there's a seamless transition of existing code to the new version of your library. Here are several things to consider when creating a new release:

Semantic Versioning

[Semantic versioning](#) (SemVer for short) is a naming convention applied to versions of your library to signify specific milestone events. Ideally, the version information you give your library should help developers determine the compatibility with their projects that make use of older versions of that same library.

The most basic approach to SemVer is the 3 component format `MAJOR.MINOR.PATCH`, where:

- `MAJOR` is incremented when you make incompatible API changes
- `MINOR` is incremented when you add functionality in a backwards-compatible manner
- `PATCH` is incremented when you make backwards-compatible bug fixes

There are also ways to specify other scenarios like pre-release versions etc. when applying version information to your .NET library.

Backwards Compatibility

As you release new versions of your library, backwards compatibility with previous versions will most likely be one of your major concerns. A new version of your library is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version. A new version of your library is binary compatible if an application that depended on the old version can, without recompilation, work with the new version.

Here are some things to consider when trying to maintain backwards compatibility with older versions of your library:

- Virtual methods: When you make a virtual method non-virtual in your new version it means that projects that override that method will have to be updated. This is a huge breaking change and is strongly discouraged.
- Method signatures: When updating a method behavior requires you to change its signature as well, you should instead create an overload so that code calling into that method will still work. You can always manipulate the old method signature to call into the new method signature so that implementation remains consistent.
- [Obsolete attribute](#): You can use this attribute in your code to specify classes or class members that are deprecated and likely to be removed in future versions. This ensures developers utilizing your library are better prepared for breaking changes.
- Optional Method Arguments: When you make previously optional method arguments compulsory or change their default value then all code that does not supply those arguments will need to be updated.

NOTE

Making compulsory arguments optional should have very little effect especially if it doesn't change the method's behavior.

The easier you make it for your users to upgrade to the new version of your library, the more likely that they will upgrade sooner.

Application Configuration File

As a .NET developer there's a very high chance you've encountered the `app.config` file present in most project types. This simple configuration file can go a long way into improving the rollout of new updates. You should generally design your libraries in such a way that information that is likely to change regularly is stored in the `app.config` file, this way when such information is updated, the config file of older versions just needs to be replaced with the new one without the need for recompilation of the library.

Consuming Libraries

As a developer that consumes .NET libraries built by other developers you're most likely aware that a new version of a library might not be fully compatible with your project and you might often find yourself having to update your code to work with those changes.

Lucky for you, C# and the .NET ecosystem comes with features and techniques that allow us to easily update our app to work with new versions of libraries that might introduce breaking changes.

Assembly Binding Redirection

You can use the `app.config` file to update the version of a library your app uses. By adding what is called a *binding redirect*, you can use the new library version without having to recompile your app. The following example shows how you would update your app's `app.config` file to use the `1.0.1` patch version of `ReferencedLibrary` instead of the `1.0.0` version it was originally compiled with.

```
<dependentAssembly>
  <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
  <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

NOTE

This approach will only work if the new version of `ReferencedLibrary` is binary compatible with your app. See the [Backwards Compatibility](#) section above for changes to look out for when determining compatibility.

new

You use the `new` modifier to hide inherited members of a base class. This is one way derived classes can respond to updates in base classes.

Take the following example:

```

public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}

```

Output

```

A base method
A derived method

```

From the example above you can see how `DerivedClass` hides the `MyMethod` method present in `BaseClass`. This means that when a base class in the new version of a library adds a member that already exists in your derived class, you can simply use the `new` modifier on your derived class member to hide the base class member.

When no `new` modifier is specified, a derived class will by default hide conflicting members in a base class, although a compiler warning will be generated the code will still compile. This means that simply adding new members to an existing class makes that new version of your library both source and binary compatible with code that depends on it.

`override`

The `override` modifier means a derived implementation extends the implementation of a base class member rather than hides it. The base class member needs to have the `virtual` modifier applied to it.

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Output

```
Base Method One: Method One
Derived Method One: Derived Method One
```

The `override` modifier is evaluated at compile time and the compiler will throw an error if it doesn't find a virtual member to override.

Your knowledge of the discussed techniques and your understanding of the situations in which to use them, will go a long way towards easing the transition between versions of a library.

How to (C#)

3/6/2021 • 3 minutes to read • [Edit Online](#)

In the How to section of the C# Guide, you can find quick answers to common questions. In some cases, articles may be listed in multiple sections. We wanted to make them easy to find for multiple search paths.

General C# concepts

There are several tips and tricks that are common C# developer practices:

- [Initialize objects using an object initializer.](#)
- [Learn the differences between passing a struct and a class to a method.](#)
- [Use operator overloading.](#)
- [Implement and call a custom extension method.](#)
- Even C# programmers may want to [use the `My` namespace from Visual Basic.](#)
- [Create a new method for an `enum` type using extension methods.](#)

Class, record, and struct members

You create classes, records, and structs to implement your program. These techniques are commonly used when writing classes, records, or structs.

- [Declare auto implemented properties.](#)
- [Declare and use read/write properties.](#)
- [Define constants.](#)
- [Override the `ToString` method to provide string output.](#)
- [Define abstract properties.](#)
- [Use the xml documentation features to document your code.](#)
- [Explicitly implement interface members](#) to keep your public interface concise.
- [Explicitly implement members of two interfaces.](#)

Working with collections

These articles help you work with collections of data.

- [Initialize a dictionary with a collection initializer.](#)

Working with strings

Strings are the fundamental data type used to display or manipulate text. These articles demonstrate common practices with strings.

- [Compare strings.](#)
- [Modify the contents of a string.](#)
- [Determine if a string represents a number.](#)
- [Use `String.Split` to separate strings.](#)
- [Combine multiple strings into one.](#)
- [Search for text in a string.](#)

Convert between types

You may need to convert an object to a different type.

- Determine if a string represents a number.
- Convert between strings that represent hexadecimal numbers and the number.
- Convert a string to a `DateTime`.
- Convert a byte array to an int.
- Convert a string to a number.
- Use pattern matching, the `as` and `is` operators to safely cast to a different type.
- Define custom type conversions.
- Determine if a type is a nullable value type.
- Convert between nullable and non-nullable value types.

Equality and ordering comparisons

You may create types that define their own rules for equality or define a natural ordering among objects of that type.

- Test for reference-based equality.
- Define value-based equality for a type.

Exception handling

.NET programs report that methods did not successfully complete their work by throwing exceptions. In these articles you'll learn to work with exceptions.

- Handle exceptions using `try` and `catch`.
- Cleanup resources using `finally` clauses.
- Recover from non-CLS (Common Language Specification) exceptions.

Delegates and events

Delegates and events provide a capability for strategies that involve loosely coupled blocks of code.

- Declare, instantiate, and use delegates.
- Combine multicast delegates.

Events provide a mechanism to publish or subscribe to notifications.

- Subscribe and unsubscribe from events.
- Implement events declared in interfaces.
- Conform to .NET guidelines when your code publishes events.
- Raise events defined in base classes from derived classes.
- Implement custom event accessors.

LINQ practices

LINQ enables you to write code to query any data source that supports the LINQ query expression pattern. These articles help you understand the pattern and work with different data sources.

- Query a collection.
- Use lambda expressions in a query.
- Use `var` in query expressions.
- Return subsets of element properties from a query.

- Write queries with complex filtering.
- Sort elements of a data source.
- Sort elements on multiple keys.
- Control the type of a projection.
- Count occurrences of a value in a source sequence.
- Calculate intermediate values.
- Merge data from multiple sources.
- Find the set difference between two sequences.
- Debug empty query results.
- Add custom methods to LINQ queries.

Multiple threads and async processing

Modern programs often use asynchronous operations. These articles will help you learn to use these techniques.

- Improve async performance using `System.Threading.Tasks.Task.WhenAll`.
- Make multiple web requests in parallel using `async` and `await`.
- Use a thread pool.

Command line args to your program

Typically, C# programs have command line arguments. These articles teach you to access and process those command line arguments.

- Retrieve all command line arguments with `for`.

How to separate strings using String.Split in C#

3/6/2021 • 2 minutes to read • [Edit Online](#)

The [String.Split](#) method creates an array of substrings by splitting the input string based on one or more delimiters. This method is often the easiest way to separate a string on word boundaries. It's also used to split strings on other specific characters or strings.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The following code splits a common phrase into an array of strings for each word.

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Every instance of a separator character produces a value in the returned array. Consecutive separator characters produce the empty string as a value in the returned array. You can see how an empty string is created in the following example, which uses the space character as a separator.

```
string phrase = "The quick brown    fox    jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

This behavior makes it easier for formats like comma-separated values (CSV) files representing tabular data. Consecutive commas represent a blank column.

You can pass an optional [StringSplitOptions.RemoveEmptyEntries](#) parameter to exclude any empty strings in the returned array. For more complicated processing of the returned collection, you can use [LINQ](#) to manipulate the result sequence.

[String.Split](#) can use multiple separator characters. The following example uses spaces, commas, periods, colons, and tabs as separating characters, which are passed to [Split](#) in an array . The loop at the bottom of the code displays each of the words in the returned array.

```

char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}

```

Consecutive instances of any separator produce the empty string in the output array:

```

char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}

```

[String.Split](#) can take an array of strings (character sequences that act as separators for parsing the target string, instead of single characters).

```

string[] separatingStrings = { "<<", "..." };

string text = "one<<two.....three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings, System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}

```

See also

- [Extract elements from a string](#)
- [C# programming guide](#)
- [Strings](#)
- [.NET regular expressions](#)

How to concatenate multiple strings (C# Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Concatenation is the process of appending one string to the end of another string. You concatenate strings by using the `+` operator. For string literals and string constants, concatenation occurs at compile time; no run-time concatenation occurs. For string variables, concatenation occurs only at run time.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The following example uses concatenation to split a long string literal into smaller strings in order to improve readability in the source code. These parts are concatenated into a single string at compile time. There is no run-time performance cost regardless of the number of strings involved.

```
// Concatenation of literals is performed at compile time, not run time.  
string text = "Historically, the world of data and the world of objects " +  
    "have not been well integrated. Programmers work in C# or Visual Basic " +  
    "and also in SQL or XQuery. On the one side are concepts such as classes, " +  
    "objects, fields, inheritance, and .NET Framework APIs. On the other side " +  
    "are tables, columns, rows, nodes, and separate languages for dealing with " +  
    "them. Data types often require translation between the two worlds; there are " +  
    "different standard functions. Because the object world has no notion of query, a " +  
    "query can only be represented as a string without compile-time type checking or " +  
    "IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +  
    "objects in memory is often tedious and error-prone.";  
  
System.Console.WriteLine(text);
```

To concatenate string variables, you can use the `+` or `+=` operators, [string interpolation](#) or the [String.Format](#), [String.Concat](#), [String.Join](#) or [StringBuilder.Append](#) methods. The `+` operator is easy to use and makes for intuitive code. Even if you use several `+` operators in one statement, the string content is copied only once. The following code shows examples of using the `+` and `+=` operators to concatenate strings:

```
string userName = "<Type your name here>";  
string dateString = DateTime.Today.ToString("dd/MM/yyyy");  
  
// Use the + and += operators for one-time concatenations.  
string str = "Hello " + userName + ". Today is " + dateString + ".  
System.Console.WriteLine(str);  
  
str += " How are you today?";  
System.Console.WriteLine(str);
```

In some expressions, it's easier to concatenate strings using string interpolation, as the following code shows:

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

NOTE

In string concatenation operations, the C# compiler treats a null string the same as an empty string.

Other method to concatenate strings is [String.Format](#). This method works well when you are building a string from a small number of component strings.

In other cases, you may be combining strings in a loop where you don't know how many source strings you're combining, and the actual number of source strings may be large. The [StringBuilder](#) class was designed for these scenarios. The following code uses the [Append](#) method of the [StringBuilder](#) class to concatenate strings.

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

You can read more about the [reasons to choose string concatenation or the `StringBuilder` class](#).

Another option to join strings from a collection is to use [String.Concat](#) method. Use [String.Join](#) method if source strings should be separated by a delimiter. The following code combines an array of words using both methods:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

At last, you can use [LINQ](#) and the [Enumerable.Aggregate](#) method to join strings from a collection. This method combines the source strings using a lambda expression. The lambda expression does the work to add each string to the existing accumulation. The following example combines an array of words by adding a space between each word in the array:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

See also

- [String](#)

- [StringBuilder](#)
- [C# programming guide](#)
- [Strings](#)

How to search strings

11/2/2020 • 4 minutes to read • [Edit Online](#)

You can use two main strategies to search for text in strings. Methods of the [String](#) class search for specific text. Regular expressions search for patterns in text.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The [string](#) type, which is an alias for the [System.String](#) class, provides a number of useful methods for searching the contents of a string. Among them are [Contains](#), [StartsWith](#), [EndsWith](#), [IndexOf](#), [LastIndexOf](#). The [System.Text.RegularExpressions.Regex](#) class provides a rich vocabulary to search for patterns in text. In this article, you learn these techniques and how to choose the best method for your needs.

Does a string contain text?

The [String.Contains](#), [String.StartsWith](#), and [String.EndsWith](#) methods search a string for specific text. The following example shows each of these methods and a variation that uses a case-insensitive search:

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
    System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult} (ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

The preceding example demonstrates an important point for using these methods. Searches are **case-sensitive** by default. You use the [StringComparison.CurrentCultureIgnoreCase](#) enumeration value to specify a case-insensitive search.

Where does the sought text occur in a string?

The [IndexOf](#) and [LastIndexOf](#) methods also search for text in strings. These methods return the location of the text being sought. If the text isn't found, they return `-1`. The following example shows a search for the first and last occurrence of the word "methods" and displays the text in between.

```

string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\": '{str2}'");

```

Finding specific text using regular expressions

The [System.Text.RegularExpressions.Regex](#) class can be used to search strings. These searches can range in complexity from simple to complicated text patterns.

The following code example searches for the word "the" or "their" in a sentence, ignoring case. The static method [Regex.IsMatch](#) performs the search. You give it the string to search and a search pattern. In this case, a third argument specifies case-insensitive search. For more information, see [System.Text.RegularExpressions.RegexOptions](#).

The search pattern describes the text you search for. The following table describes each element of the search pattern. (The table below uses the single `\`, which must be escaped as `\\\` in a C# string).

PATTERN	MEANING
<code>the</code>	match the text "the"
<code>(eir)?</code>	match 0 or 1 occurrence of "eir"
<code>\s</code>	match a white-space character

```

string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.WriteLine($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}

```

TIP

The `string` methods are usually better choices when you are searching for an exact string. Regular expressions are better when you are searching for some pattern in a source string.

Does a string follow a pattern?

The following code uses regular expressions to validate the format of each string in an array. The validation requires that each string have the form of a telephone number in which three groups of digits are separated by dashes, the first two groups contain three digits, and the third group contains four digits. The search pattern uses the regular expression `^\d{3}-\d{3}-\d{4}$`. For more information, see [Regular Expression Language - Quick Reference](#).

PATTERN	MEANING
<code>^</code>	matches the beginning of the string
<code>\d{3}</code>	matches exactly 3 digit characters
<code>-</code>	matches the '-' character
<code>\d{3}</code>	matches exactly 3 digit characters
<code>-</code>	matches the '-' character
<code>\d{4}</code>	matches exactly 4 digit characters
<code>\$</code>	matches the end of the string

```

string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.WriteLine($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}

```

This single search pattern matches many valid strings. Regular expressions are better to search for or validate against a pattern, rather than a single text string.

See also

- [C# programming guide](#)
- [Strings](#)
- [LINQ and strings](#)
- [System.Text.RegularExpressions.Regex](#)
- [.NET regular expressions](#)
- [Regular expression language - quick reference](#)
- [Best practices for using strings in .NET](#)

How to modify string contents in C#

11/2/2020 • 5 minutes to read • [Edit Online](#)

This article demonstrates several techniques to produce a `string` by modifying an existing `string`. All the techniques demonstrated return the result of the modifications as a new `string` object. To demonstrate that the original and modified strings are distinct instances, the examples store the result in a new variable. You can examine the original `string` and the new, modified `string` when you run each example.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

There are several techniques demonstrated in this article. You can replace existing text. You can search for patterns and replace matching text with other text. You can treat a string as a sequence of characters. You can also use convenience methods that remove white space. Choose the techniques that most closely match your scenario.

Replace text

The following code creates a new string by replacing existing text with a substitute.

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

The preceding code demonstrates this *immutable* property of strings. You can see in the preceding example that the original string, `source`, is not modified. The [String.Replace](#) method creates a new `string` containing the modifications.

The [Replace](#) method can replace either strings or single characters. In both cases, every occurrence of the sought text is replaced. The following example replaces all ' ' characters with '_':

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

The source string is unchanged, and a new string is returned with the replacement.

Trim white space

You can use the [String.Trim](#), [String.TrimStart](#), and [String.TrimEnd](#) methods to remove any leading or trailing white

space. The following code shows an example of each. The source string does not change; these methods return a new string with the modified contents.

```
// Remove trailing and leading white space.  
string source = "    I'm wider than I need to be.      ";  
// Store the results in a new string variable.  
var trimmedResult = source.Trim();  
var trimLeading = source.TrimStart();  
var trimTrailing = source.TrimEnd();  
Console.WriteLine($"<{source}>");  
Console.WriteLine($"<{trimmedResult}>");  
Console.WriteLine($"<{trimLeading}>");  
Console.WriteLine($"<{trimTrailing}>");
```

Remove text

You can remove text from a string using the [String.Remove](#) method. This method removes a number of characters starting at a specific index. The following example shows how to use [String.IndexOf](#) followed by [Remove](#) to remove text from a string:

```
string source = "Many mountains are behind many clouds today.";  
// Remove a substring from the middle of the string.  
string toRemove = "many ";  
string result = string.Empty;  
int i = source.IndexOf(toRemove);  
if (i >= 0)  
{  
    result = source.Remove(i, toRemove.Length);  
}  
Console.WriteLine(source);  
Console.WriteLine(result);
```

Replace matching patterns

You can use [regular expressions](#) to replace text matching patterns with new text, possibly defined by a pattern. The following example uses the [System.Text.RegularExpressions.Regex](#) class to find a pattern in a source string and replace it with proper capitalization. The [Regex.Replace\(String, String, MatchEvaluator, RegexOptions\)](#) method takes a function that provides the logic of the replacement as one of its arguments. In this example, that function, `LocalReplaceMatchCase` is a **local function** declared inside the sample method. `LocalReplaceMatchCase` uses the [System.Text.StringBuilder](#) class to build the replacement string with proper capitalization.

Regular expressions are most useful for searching and replacing text that follows a pattern, rather than known text. For more information, see [How to search strings](#). The search pattern, "the\s" searches for the word "the" followed by a white-space character. That part of the pattern ensures that it doesn't match "there" in the source string. For more information on regular expression language elements, see [Regular Expression Language - Quick Reference](#).

```

string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s", LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}

```

The [StringBuilder.ToString](#) method returns an immutable string with the contents in the [StringBuilder](#) object.

Modifying individual characters

You can produce a character array from a string, modify the contents of the array, and then create a new string from the modified contents of the array.

The following example shows how to replace a set of characters in a string. First, it uses the [String.ToCharArray\(\)](#) method to create an array of characters. It uses the [IndexOf](#) method to find the starting index of the word "fox." The next three characters are replaced with a different word. Finally, a new string is constructed from the updated character array.

```

string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);

```

Programmatically build up string content

Since strings are immutable, the previous examples all create temporary strings or character arrays. In high-performance scenarios, it may be desirable to avoid these heap allocations. .NET Core provides a [String.Create](#) method that allows you to programmatically fill in the character content of a string via a callback while avoiding the intermediate temporary string allocations.

```
// constructing a string from a char array, prefix it with some additional characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[] charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

You could modify a string in a fixed block with unsafe code, but it is **strongly** discouraged to modify the string content after a string is created. Doing so will break things in unpredictable ways. For example, if someone interns a string that has the same content as yours, they'll get your copy and won't expect that you are modifying their string.

See also

- [.NET regular expressions](#)
- [Regular expression language - quick reference](#)

How to compare strings in C#

3/6/2021 • 11 minutes to read • [Edit Online](#)

You compare strings to answer one of two questions: "Are these two strings equal?" or "In what order should these strings be placed when sorting them?"

Those two questions are complicated by factors that affect string comparisons:

- You can choose an ordinal or linguistic comparison.
- You can choose if case matters.
- You can choose culture-specific comparisons.
- Linguistic comparisons are culture and platform-dependent.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

When you compare strings, you define an order among them. Comparisons are used to sort a sequence of strings. Once the sequence is in a known order, it is easier to search, both for software and for humans. Other comparisons may check if strings are the same. These sameness checks are similar to equality, but some differences, such as case differences, may be ignored.

Default ordinal comparisons

By default, the most common operations:

- [String.Equals](#)
- [String.Equality](#) and [String.Inequality](#), that is, equality operators `==` and `!=`, respectively

perform a case-sensitive ordinal comparison and, if necessary, use the current culture. The following example demonstrates that:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not equal")});
```

The default ordinal comparison doesn't take linguistic rules into account when comparing strings. It compares the binary value of each [Char](#) object in two strings. As a result, the default ordinal comparison is also case-sensitive.

The test for equality with [String.Equals](#) and the `==` and `!=` operators differs from string comparison using the

`String.CompareTo` and `Compare(String, String)` methods. While the tests for equality perform a case-sensitive ordinal comparison, the comparison methods perform a case-sensitive, culture-sensitive comparison using the current culture. Because the default comparison methods often perform different types of comparisons, we recommend that you always make the intent of your code clear by calling an overload that explicitly specifies the type of comparison to perform.

Case-insensitive ordinal comparisons

The `String.Equals(String, StringComparison)` method enables you to specify a `StringComparison` value of `StringComparison.OrdinalIgnoreCase` for a case-insensitive ordinal comparison. There is also a static `String.Compare(String, String, StringComparison)` method that performs a case-insensitive ordinal comparison if you specify a value of `StringComparison.OrdinalIgnoreCase` for the `StringComparison` argument. These are shown in the following code:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2, StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are {(areEqual ? "equal." : "not equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

When performing a case-insensitive ordinal comparison, these methods use the casing conventions of the [invariant culture](#).

Linguistic comparisons

Strings can also be ordered using linguistic rules for the current culture. This is sometimes referred to as "word sort order." When you perform a linguistic comparison, some nonalphanumeric Unicode characters might have special weights assigned. For example, the hyphen "-" may have a small weight assigned to it so that "co-op" and "coop" appear next to each other in sort order. In addition, some Unicode characters may be equivalent to a sequence of `Char` instances. The following example uses the phrase "They dance in the street." in German with the "ss" (U+0073 U+0073) in one string and 'ß' (U+00DF) in another. Linguistically (in Windows), "ss" is equal to the German Esszet: 'ß' character in both the "en-US" and "de-DE" cultures.

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second, StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")} equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two, StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}

```

This sample demonstrates the operating system-dependent nature of linguistic comparisons. The host for the interactive window is a Linux host. The linguistic and ordinal comparisons produce the same results. If you run this same sample on a Windows host, you see the following output:

```

<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison

```

On Windows, the sort order of "cop", "coop", and "co-op" change when you change from a linguistic comparison to an ordinal comparison. The two German sentences also compare differently using the different comparison types.

Comparisons using specific cultures

This sample stores [CultureInfo](#) objects for the en-US and de-DE cultures. The comparisons are performed using a [CultureInfo](#) object to ensure a culture-specific comparison.

The culture used affects linguistic comparisons. The following example shows the results of comparing the two German sentences using the "en-US" culture and the "de-DE" culture:

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo culture)
{
    int compareLinguistic = String.Compare(one, two, en, System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}

```

Culture-sensitive comparisons are typically used to compare and sort strings input by users with other strings input by users. The characters and sorting conventions of these strings might vary depending on the locale of the user's computer. Even strings that contain identical characters might sort differently depending on the culture of the current thread. In addition, try this sample code locally on a Windows machine, and you'll get the following results:

```

<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison

```

Linguistic comparisons are dependent on the current culture, and are OS dependent. Take that into account when you work with string comparisons.

Linguistic sorting and searching strings in arrays

The following examples show how to sort and search for strings in an array using a linguistic comparison dependent on the current culture. You use the static [Array](#) methods that take a [System.StringComparer](#) parameter.

This example shows how to sort an array of strings using the current culture:

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}
```

Once the array is sorted, you can search for entries using a binary search. A binary search starts in the middle of the collection to determine which half of the collection would contain the sought string. Each subsequent comparison subdivides the remaining part of the collection in half. The array is sorted using the [StringComparer.CurrentCulture](#). The local function `ShowWhere` displays information about where the string was found. If the string wasn't found, the returned value indicates where it would be if it were found.

```

string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString, StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{{(result > 0 ? "Found" : "Did not find")}} {searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.WriteLine($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}

```

Ordinal sorting and searching in collections

The following code uses the [System.Collections.Generic.List<T>](#) collection class to store strings. The strings are sorted using the [List<T>.Sort](#) method. This method needs a delegate that compares and orders two strings. The [String.CompareTo](#) method provides that comparison function. Run the sample and observe the order. This sort operation uses an ordinal case-sensitive sort. You would use the static [String.Compare](#) methods to specify different comparison rules.

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}
```

Once sorted, the list of strings can be searched using a binary search. The following sample shows how to search the sorted listed using the same comparison function. The local function `ShowWhere` shows where the sought text is or would be:

```

List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{{(result > 0 ? "Found" : "Did not find")}} {searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}

```

Always make sure to use the same type of comparison for sorting and searching. Using different comparison types for sorting and searching produces unexpected results.

Collection classes such as [System.Collections.Hashtable](#), [System.Collections.Generic.Dictionary< TKey, TValue >](#), and [System.Collections.Generic.List< T >](#) have constructors that take a [System.StringComparer](#) parameter when the type of the elements or keys is `string`. In general, you should use these constructors whenever possible, and specify either [StringComparer.Ordinal](#) or [StringComparer.OrdinalIgnoreCase](#).

Reference equality and string interning

None of the samples have used [ReferenceEquals](#). This method determines if two strings are the same object, which can lead to inconsistent results in string comparisons. The following example demonstrates the *string interning* feature of C#. When a program declares two or more identical string variables, the compiler stores them all in the same location. By calling the [ReferenceEquals](#) method, you can see that the two strings actually refer to the same object in memory. Use the [String.Copy](#) method to avoid interning. After the copy has been made, the two strings have different storage locations, even though they have the same value. Run the following sample to show that strings `a` and `b` are *interned* meaning they share the same storage. The strings `a` and `c` are not.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

When you test for equality of strings, you should use the methods that explicitly specify what kind of comparison you intend to perform. Your code is much more maintainable and readable. Use the overloads of the methods of the [System.String](#) and [System.Array](#) classes that take a [StringComparison](#) enumeration parameter. You specify which type of comparison to perform. Avoid using the `==` and `!=` operators when you test for equality. The [String.CompareTo](#) instance methods always perform an ordinal case-sensitive comparison. They are primarily suited for ordering strings alphabetically.

You can intern a string or retrieve a reference to an existing interned string by calling the [String.Intern](#) method. To determine whether a string is interned, call the [String.IsInterned](#) method.

See also

- [System.Globalization.CultureInfo](#)
- [System.StringComparer](#)
- [Strings](#)
- [Comparing strings](#)
- [Globalizing and localizing applications](#)

How to safely cast by using pattern matching and the is and as operators

11/2/2020 • 4 minutes to read • [Edit Online](#)

Because objects are polymorphic, it's possible for a variable of a base class type to hold a derived [type](#). To access the derived type's instance members, it's necessary to [cast](#) the value back to the derived type. However, a cast creates the risk of throwing an [InvalidOperationException](#). C# provides [pattern matching](#) statements that perform a cast conditionally only when it will succeed. C# also provides the [is](#) and [as](#) operators to test if a value is of a certain type.

The following example shows how to use the pattern matching `is` statement:

```

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        var g = new Giraffe();
        var a = new Animal();
        FeedMammals(g);
        FeedMammals(a);
        // Output:
        // Eating.
        // Animal is not a Mammal

        SuperNova sn = new SuperNova();
        TestForMammals(g);
        TestForMammals(sn);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    static void FeedMammals(Animal a)
    {
        if (a is Mammal m)
        {
            m.Eat();
        }
        else
        {
            // variable 'm' is not in scope here, and can't be used.
            Console.WriteLine($"{a.GetType().Name} is not a Mammal");
        }
    }

    static void TestForMammals(object o)
    {
        // You also can use the as operator and test for null
        // before referencing the variable.
        var m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }
}

```

The preceding sample demonstrates a few features of pattern matching syntax. The `if (a is Mammal m)` statement combines the test with an initialization assignment. The assignment occurs only when the test succeeds. The variable `m` is only in scope in the embedded `if` statement where it has been assigned. You

cannot access `m` later in the same method. The preceding example also shows how to use the `as` operator to convert an object to a specified type.

You can also use the same syntax for testing if a nullable value type has a value, as shown in the following example:

```

class Program
{
    static void Main(string[] args)
    {
        int i = 5;
        PatternMatchingNullable(i);

        int? j = null;
        PatternMatchingNullable(j);

        double d = 9.78654;
        PatternMatchingNullable(d);

        PatternMatchingSwitch(i);
        PatternMatchingSwitch(j);
        PatternMatchingSwitch(d);
    }

    static void PatternMatchingNullable(System.ValueType val)
    {
        if (val is int j) // Nullable types are not allowed in patterns
        {
            Console.WriteLine(j);
        }
        else if (val is null) // If val is a nullable type with no value, this expression is true
        {
            Console.WriteLine("val is a nullable type with the null value");
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }

    static void PatternMatchingSwitch(System.ValueType val)
    {
        switch (val)
        {
            case int number:
                Console.WriteLine(number);
                break;
            case long number:
                Console.WriteLine(number);
                break;
            case decimal number:
                Console.WriteLine(number);
                break;
            case float number:
                Console.WriteLine(number);
                break;
            case double number:
                Console.WriteLine(number);
                break;
            case null:
                Console.WriteLine("val is a nullable type with the null value");
                break;
            default:
                Console.WriteLine("Could not convert " + val.ToString());
                break;
        }
    }
}

```

The preceding sample demonstrates other features of pattern matching to use with conversions. You can test a variable for the null pattern by checking specifically for the `null` value. When the runtime value of the variable is `null`, an `is` statement checking for a type always returns `false`. The pattern matching `is` statement

doesn't allow a nullable value type, such as `int?` or `Nullable<int>`, but you can test for any other value type. The `is` patterns from the preceding example are not limited to the nullable value types. You can also use those patterns to test if a variable of a reference type has a value or it's `null`.

The preceding sample also shows how you use the type pattern in a `switch` statement where the variable may be one of many different types.

If you want to test if a variable is a given type, but not assign it to a new variable, you can use the `is` and `as` operators for reference types and nullable value types. The following code shows how to use the `is` and `as` statements that were part of the C# language before pattern matching was introduced to test if a variable is of a given type:

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        UseAsOperator(sn);

        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        UseAsWithNullable(i);

        double d = 9.78654;
        UseAsWithNullable(d);
    }

    static void UseIsOperator(Animal a)
    {
        if (a is Mammal)
        {
            Mammal m = (Mammal)a;
            m.Eat();
        }
    }

    static void UsePatternMatchingIs(Animal a)
    {
        if (a is Mammal m)
        {
            m.Eat();
        }
    }
}
```

```

        {
            m.Eat();
        }
    }

    static void UseAsOperator(object o)
    {
        Mammal m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }

    static void UseAsWithNullable(System.ValueType val)
    {
        int? j = val as int?;
        if (j != null)
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }
}

```

As you can see by comparing this code with the pattern matching code, the pattern matching syntax provides more robust features by combining the test and the assignment in a single statement. Use the pattern matching syntax whenever possible.

The .NET Compiler Platform SDK

3/6/2021 • 6 minutes to read • [Edit Online](#)

Compilers build a detailed model of application code as they validate the syntax and semantics of that code. They use this model to build the executable output from the source code. The .NET Compiler Platform SDK provides access to this model. Increasingly, we rely on integrated development environment (IDE) features such as IntelliSense, refactoring, intelligent rename, "Find all references," and "Go to definition" to increase our productivity. We rely on code analysis tools to improve our code quality, and code generators to aid in application construction. As these tools get smarter, they need access to more and more of the model that only compilers create as they process application code. This is the core mission of the Roslyn APIs: opening up the opaque boxes and allowing tools and end users to share in the wealth of information compilers have about our code. Instead of being opaque source-code-in and object-code-out translators, through Roslyn, compilers become platforms: APIs that you can use for code-related tasks in your tools and applications.

.NET Compiler Platform SDK concepts

The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and Visual Basic languages, and embedding of C# and Visual Basic in domain-specific languages.

The .NET Compiler Platform SDK enables you to build *analyzers* and *code fixes* that find and correct coding mistakes. *Analyzers* understand the syntax (structure of code) and semantics to detect practices that should be corrected. *Code fixes* provide one or more suggested fixes for addressing coding mistakes found by analyzers or compiler diagnostics. Typically, an analyzer and the associated code fixes are packaged together in a single project.

Analyzers and code fixes use static analysis to understand code. They do not run the code or provide other testing benefits. They can, however, point out practices that often lead to bugs, unmaintainable code, or standard guideline violation.

In addition to analyzers and code fixes, The .NET Compiler Platform SDK also enables you to build *code refactorings*. It also provides a single set of APIs that enable you to examine and understand a C# or Visual Basic codebase. Because you can use this single codebase, you can write analyzers and code fixes more easily by leveraging the syntactic and semantic analysis APIs provided by the .NET Compiler Platform SDK. Freed from the large task of replicating the analysis done by the compiler, you can concentrate on the more focused task of finding and fixing common coding errors for your project or library.

A smaller benefit is that your analyzers and code fixes are smaller and use much less memory when loaded in Visual Studio than they would if you wrote your own codebase to understand the code in a project. By leveraging the same classes used by the compiler and Visual Studio, you can create your own static analysis tools. This means your team can use analyzers and code fixes without a noticeable impact on the IDE's performance.

There are three main scenarios for writing analyzers and code fixes:

1. *Enforce team coding standards*
2. *Provide guidance with library packages*
3. *Provide general guidance*

Enforce team coding standards

Many teams have coding standards that are enforced through code reviews with other team members. Analyzers and code fixes can make this process much more efficient. Code reviews happen when a developer shares their work with others on the team. The developer will have invested all the time needed to complete a new feature before getting any comments. Weeks may go by while the developer reinforces habits that don't match the team's practices.

Analyzers run as a developer writes code. The developer gets immediate feedback that encourages following the guidance immediately. The developer builds habits to write compliant code as soon as they begin prototyping. When the feature is ready for humans to review, all the standard guidance has been enforced.

Teams can build analyzers and code fixes that look for the most common practices that violate team coding practices. These can be installed on each developer's machine to enforce the standards.

TIP

Before building your own analyzer, check out the built-in ones. For more information, see [Code-style rules](#).

Provide guidance with library packages

There is a wealth of libraries available for .NET developers on NuGet. Some of these come from Microsoft, some from third-party companies, and others from community members and volunteers. These libraries get more adoption and higher reviews when developers can succeed with those libraries.

In addition to providing documentation, you can provide analyzers and code fixes that find and correct common mis-uses of your library. These immediate corrections will help developers succeed more quickly.

You can package analyzers and code fixes with your library on NuGet. In that scenario, every developer who installs your NuGet package will also install the analyzer package. All developers using your library will immediately get guidance from your team in the form of immediate feedback on mistakes and suggested corrections.

Provide general guidance

The .NET developer community has discovered, through experience, patterns that work well and patterns that are best avoided. Several community members have created analyzers that enforce those recommended patterns. As we learn more, there is always room for new ideas.

These analyzers can be uploaded to the [Visual Studio Marketplace](#) and downloaded by developers using Visual Studio. Newcomers to the language and the platform learn accepted practices quickly and become productive earlier in their .NET journey. As these become more widely used, the community adopts these practices.

Next steps

The .NET Compiler Platform SDK includes the latest language object models for code generation, analysis, and refactoring. This section provides a conceptual overview of the .NET Compiler Platform SDK. Further details can be found in the quickstarts, samples, and tutorials sections.

You can learn more about the concepts in the .NET Compiler Platform SDK in these five topics:

- [Explore code with the syntax visualizer](#)
- [Understand the compiler API model](#)
- [Work with syntax](#)
- [Work with semantics](#)
- [Work with a workspace](#)

To get started, you'll need to install the .NET Compiler Platform SDK:

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the Visual Studio Installer:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

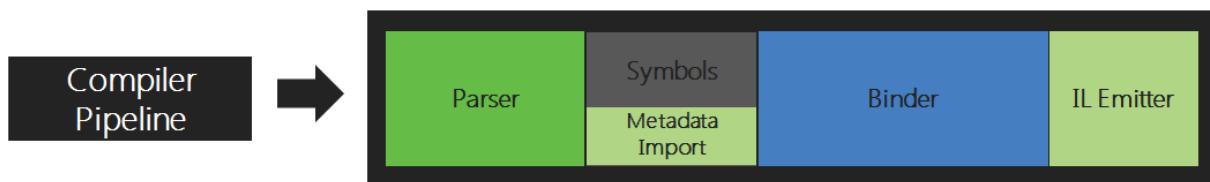
Understand the .NET Compiler Platform SDK model

3/6/2021 • 3 minutes to read • [Edit Online](#)

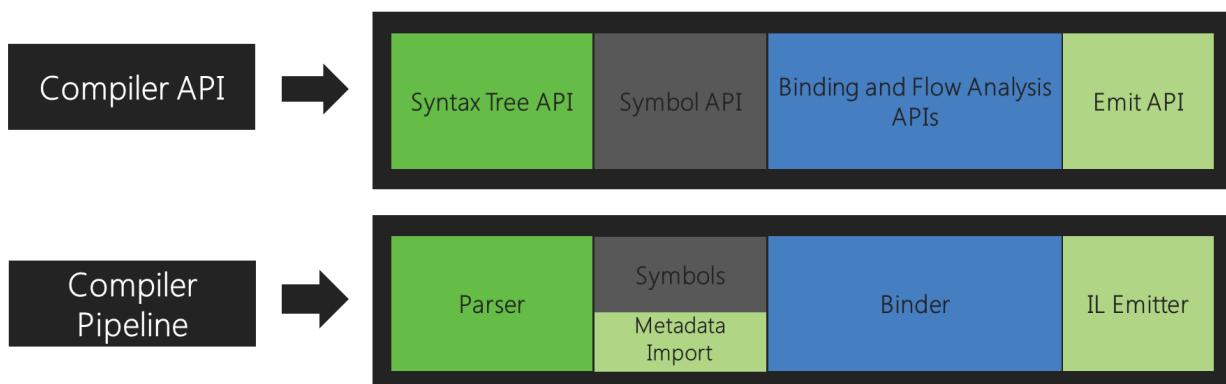
Compilers process the code you write following structured rules that often differ from the way humans read and understand code. A basic understanding of the model used by compilers is essential to understanding the APIs you use when building Roslyn-based tools.

Compiler pipeline functional areas

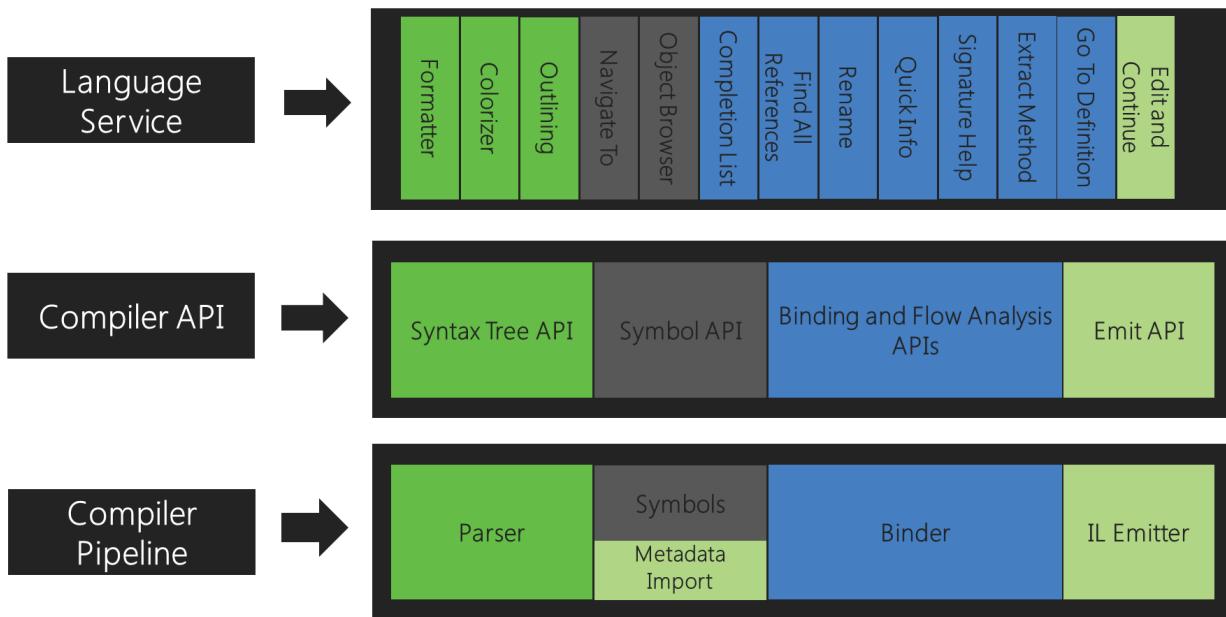
The .NET Compiler Platform SDK exposes the C# and Visual Basic compilers' code analysis to you as a consumer by providing an API layer that mirrors a traditional compiler pipeline.



Each phase of this pipeline is a separate component. First, the parse phase tokenizes and parses source text into syntax that follows the language grammar. Second, the declaration phase analyzes source and imported metadata to form named symbols. Next, the bind phase matches identifiers in the code to symbols. Finally, the emit phase emits an assembly with all the information built up by the compiler.



Corresponding to each of those phases, the .NET Compiler Platform SDK exposes an object model that allows access to the information at that phase. The parsing phase exposes a syntax tree, the declaration phase exposes a hierarchical symbol table, the binding phase exposes the result of the compiler's semantic analysis, and the emit phase is an API that produces IL byte codes.



Each compiler combines these components together as a single end-to-end whole.

These APIs are the same ones used by Visual Studio. For instance, the code outlining and formatting features use the syntax trees, the **Object Browser**, and navigation features use the symbol table, refactorings and **Go to Definition** use the semantic model, and **Edit and Continue** uses all of these, including the Emit API.

API layers

The .NET compiler SDK consists of several layers of APIs: compiler APIs, diagnostic APIs, scripting APIs, and workspaces APIs.

Compiler APIs

The compiler layer contains the object models that correspond to information exposed at each phase of the compiler pipeline, both syntactic and semantic. The compiler layer also contains an immutable snapshot of a single invocation of a compiler, including assembly references, compiler options, and source code files. There are two distinct APIs that represent the C# language and the Visual Basic language. The two APIs are similar in shape but tailored for high-fidelity to each individual language. This layer has no dependencies on Visual Studio components.

Diagnostic APIs

As part of its analysis, the compiler may produce a set of diagnostics covering everything from syntax, semantic, and definite assignment errors to various warnings and informational diagnostics. The Compiler API layer exposes diagnostics through an extensible API that allows user-defined analyzers to be plugged into the compilation process. It allows user-defined diagnostics, such as those produced by tools like StyleCop, to be produced alongside compiler-defined diagnostics. Producing diagnostics in this way has the benefit of integrating naturally with tools such as MSBuild and Visual Studio, which depend on diagnostics for experiences such as halting a build based on policy and showing live squiggles in the editor and suggesting code fixes.

Scripting APIs

Hosting and scripting APIs are part of the compiler layer. You can use them for executing code snippets and accumulating a runtime execution context. The C# interactive REPL (Read-Evaluate-Print Loop) uses these APIs. The REPL enables you to use C# as a scripting language, executing the code interactively as you write it.

Workspaces APIs

The Workspaces layer contains the Workspace API, which is the starting point for doing code analysis and refactoring over entire solutions. It assists you in organizing all the information about the projects in a solution into a single object model, offering you direct access to the compiler layer object models without needing to

parse files, configure options, or manage project-to-project dependencies.

In addition, the Workspaces layer surfaces a set of APIs used when implementing code analysis and refactoring tools that function within a host environment like the Visual Studio IDE. Examples include the Find All References, Formatting, and Code Generation APIs.

This layer has no dependencies on Visual Studio components.

Work with syntax

3/6/2021 • 8 minutes to read • [Edit Online](#)

The *syntax tree* is a fundamental immutable data structure exposed by the compiler APIs. These trees represent the lexical and syntactic structure of source code. They serve two important purposes:

- To allow tools - such as an IDE, add-ins, code analysis tools, and refactorings - to see and process the syntactic structure of source code in a user's project.
- To enable tools - such as refactorings and an IDE - to create, modify, and rearrange source code in a natural manner without having to use direct text edits. By creating and manipulating trees, tools can easily create and rearrange source code.

Syntax trees

Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation. No part of the source code is understood without it first being identified and categorized into one of many well-known structural language elements.

Syntax trees have three key attributes:

- They hold all the source information in full fidelity. Full fidelity means that the syntax tree contains every piece of information found in the source text, every grammatical construct, every lexical token, and everything else in between, including white space, comments, and preprocessor directives. For example, each literal mentioned in the source is represented exactly as it was typed. Syntax trees also capture errors in source code when the program is incomplete or malformed by representing skipped or missing tokens.
- They can produce the exact text that they were parsed from. From any syntax node, it's possible to get the text representation of the subtree rooted at that node. This ability means that syntax trees can be used as a way to construct and edit source text. By creating a tree you have, by implication, created the equivalent text, and by making a new tree out of changes to an existing tree, you have effectively edited the text.
- They are immutable and thread-safe. After a tree is obtained, it's a snapshot of the current state of the code and never changes. This allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. Because the trees are immutable and no modifications can be made directly to a tree, factory methods help create and modify syntax trees by creating additional snapshots of the tree. The trees are efficient in the way they reuse underlying nodes, so a new version can be rebuilt fast and with little extra memory.

A syntax tree is literally a tree data structure, where non-terminal structural elements parent other elements. Each syntax tree is made up of nodes, tokens, and trivia.

Syntax nodes

Syntax nodes are one of the primary elements of syntax trees. These nodes represent syntactic constructs such as declarations, statements, clauses, and expressions. Each category of syntax nodes is represented by a separate class derived from [Microsoft.CodeAnalysis.SyntaxNode](#). The set of node classes is not extensible.

All syntax nodes are non-terminal nodes in the syntax tree, which means they always have other nodes and tokens as children. As a child of another node, each node has a parent node that can be accessed through the [SyntaxNode.Parent](#) property. Because nodes and trees are immutable, the parent of a node never changes. The root of the tree has a null parent.

Each node has a [SyntaxNode.ChildNodes\(\)](#) method, which returns a list of child nodes in sequential order based

on their position in the source text. This list does not contain tokens. Each node also has methods to examine Descendants, such as [DescendantNodes](#), [DescendantTokens](#), or [DescendantTrivia](#) - that represent a list of all the nodes, tokens, or trivia that exist in the subtree rooted by that node.

In addition, each syntax node subclass exposes all the same children through strongly typed properties. For example, a [BinaryExpressionSyntax](#) node class has three additional properties specific to binary operators: [Left](#), [OperatorToken](#), and [Right](#). The type of [Left](#) and [Right](#) is [ExpressionSyntax](#), and the type of [OperatorToken](#) is [SyntaxToken](#).

Some syntax nodes have optional children. For example, an [IfStatementSyntax](#) has an optional [ElseClauseSyntax](#). If the child is not present, the property returns null.

Syntax tokens

Syntax tokens are the terminals of the language grammar, representing the smallest syntactic fragments of the code. They are never parents of other nodes or tokens. Syntax tokens consist of keywords, identifiers, literals, and punctuation.

For efficiency purposes, the [SyntaxToken](#) type is a CLR value type. Therefore, unlike syntax nodes, there is only one structure for all kinds of tokens with a mix of properties that have meaning depending on the kind of token that is being represented.

For example, an integer literal token represents a numeric value. In addition to the raw source text the token spans, the literal token has a [Value](#) property that tells you the exact decoded integer value. This property is typed as [Object](#) because it may be one of many primitive types.

The [ValueText](#) property tells you the same information as the [Value](#) property; however this property is always typed as [String](#). An identifier in C# source text may include Unicode escape characters, yet the syntax of the escape sequence itself is not considered part of the identifier name. So although the raw text spanned by the token does include the escape sequence, the [ValueText](#) property does not. Instead, it includes the Unicode characters identified by the escape. For example, if the source text contains an identifier written as `\u03c0`, then the [ValueText](#) property for this token will return π .

Syntax trivia

Syntax trivia represent the parts of the source text that are largely insignificant for normal understanding of the code, such as white space, comments, and preprocessor directives. Like syntax tokens, trivia are value types. The single [Microsoft.CodeAnalysis.SyntaxTrivia](#) type is used to describe all kinds of trivia.

Because trivia are not part of the normal language syntax and can appear anywhere between any two tokens, they are not included in the syntax tree as a child of a node. Yet, because they are important when implementing a feature like refactoring and to maintain full fidelity with the source text, they do exist as part of the syntax tree.

You can access trivia by inspecting a token's [SyntaxToken.LeadingTrivia](#) or [SyntaxToken.TrailingTrivia](#) collections. When source text is parsed, sequences of trivia are associated with tokens. In general, a token owns any trivia after it on the same line up to the next token. Any trivia after that line is associated with the following token. The first token in the source file gets all the initial trivia, and the last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

Unlike syntax nodes and tokens, syntax trivia do not have parents. Yet, because they are part of the tree and each is associated with a single token, you may access the token it is associated with using the [SyntaxTrivia.Token](#) property.

Spans

Each node, token, or trivia knows its position within the source text and the number of characters it consists of. A

text position is represented as a 32-bit integer, which is a zero-based `char` index. A `TextSpan` object is the beginning position and a count of characters, both represented as integers. If `TextSpan` has a zero length, it refers to a location between two characters.

Each node has two `TextSpan` properties: `Span` and `FullSpan`.

The `Span` property is the text span from the start of the first token in the node's subtree to the end of the last token. This span does not include any leading or trailing trivia.

The `FullSpan` property is the text span that includes the node's normal span, plus the span of any leading or trailing trivia.

For example:

```
if (x > 3)
{
||      // this is bad
|throw new Exception("Not right.");|  // better exception?||
}
```

The statement node inside the block has a span indicated by the single vertical bars (`||`). It includes the characters `throw new Exception("Not right.");`. The full span is indicated by the double vertical bars (`|||`). It includes the same characters as the span and the characters associated with the leading and trailing trivia.

Kinds

Each node, token, or trivia has a `SyntaxNode.RawKind` property, of type `System.Int32`, that identifies the exact syntax element represented. This value can be cast to a language-specific enumeration. Each language, C# or Visual Basic, has a single `SyntaxKind` enumeration (`Microsoft.CodeAnalysis.CSharp.SyntaxKind` and `Microsoft.CodeAnalysis.VisualBasic.SyntaxKind`, respectively) that lists all the possible nodes, tokens, and trivia elements in the grammar. This conversion can be done automatically by accessing the `CSharpExtensions.Kind` or `VisualBasicExtensions.Kind` extension methods.

The `RawKind` property allows for easy disambiguation of syntax node types that share the same node class. For tokens and trivia, this property is the only way to distinguish one type of element from another.

For example, a single `BinaryExpressionSyntax` class has `Left`, `OperatorToken`, and `Right` as children. The `Kind` property distinguishes whether it is an `AddExpression`, `SubtractExpression`, or `MultiplyExpression` kind of syntax node.

TIP

It's recommended to check kinds using `IsKind` (for C#) or `IsKind` (for VB) extension methods.

Errors

Even when the source text contains syntax errors, a full syntax tree that is round-trippable to the source is exposed. When the parser encounters code that does not conform to the defined syntax of the language, it uses one of two techniques to create a syntax tree:

- If the parser expects a particular kind of token but does not find it, it may insert a missing token into the syntax tree in the location that the token was expected. A missing token represents the actual token that was expected, but it has an empty span, and its `SyntaxNode.IsMissing` property returns `true`.
- The parser may skip tokens until it finds one where it can continue parsing. In this case, the skipped tokens are attached as a trivia node with the kind `SkippedTokensTrivia`.

Work with semantics

3/6/2021 • 3 minutes to read • [Edit Online](#)

[Syntax trees](#) represent the lexical and syntactic structure of source code. Although this information alone is enough to describe all the declarations and logic in the source, it is not enough information to identify what is being referenced. A name may represent:

- a type
- a field
- a method
- a local variable

Although each of these is uniquely different, determining which one an identifier actually refers to often requires a deep understanding of the language rules.

There are program elements represented in source code, and programs can also refer to previously compiled libraries, packaged in assembly files. Although no source code, and therefore no syntax nodes or trees, are available for assemblies, programs can still refer to elements inside them.

For those tasks, you need the [Semantic model](#).

In addition to a syntactic model of the source code, a semantic model encapsulates the language rules, giving you an easy way to correctly match identifiers with the correct program element being referenced.

Compilation

A compilation is a representation of everything needed to compile a C# or Visual Basic program, which includes all the assembly references, compiler options, and source files.

Because all this information is in one place, the elements contained in the source code can be described in more detail. The compilation represents each declared type, member, or variable as a symbol. The compilation contains a variety of methods that help you find and relate the symbols that have either been declared in the source code or imported as metadata from an assembly.

Similar to syntax trees, compilations are immutable. After you create a compilation, it cannot be changed by you or anyone else you might be sharing it with. However, you can create a new compilation from an existing compilation, specifying a change as you do so. For example, you might create a compilation that is the same in every way as an existing compilation, except it may include an additional source file or assembly reference.

Symbols

A symbol represents a distinct element declared by the source code or imported from an assembly as metadata. Every namespace, type, method, property, field, event, parameter, or local variable is represented by a symbol.

A variety of methods and properties on the [Compilation](#) type help you find symbols. For example, you can find a symbol for a declared type by its common metadata name. You can also access the entire symbol table as a tree of symbols rooted by the global namespace.

Symbols also contain additional information that the compiler determines from the source or metadata, such as other referenced symbols. Each kind of symbol is represented by a separate interface derived from [ISymbol](#), each with its own methods and properties detailing the information the compiler has gathered. Many of these properties directly reference other symbols. For example, the [IMethodSymbol.ReturnType](#) property tells you the

actual type symbol that the method returns.

Symbols present a common representation of namespaces, types, and members, between source code and metadata. For example, a method that was declared in source code and a method that was imported from metadata are both represented by an [IMethodSymbol](#) with the same properties.

Symbols are similar in concept to the CLR type system as represented by the [System.Reflection](#) API, yet they are richer in that they model more than just types. Namespaces, local variables, and labels are all symbols. In addition, symbols are a representation of language concepts, not CLR concepts. There is a lot of overlap, but there are many meaningful distinctions as well. For instance, an iterator method in C# or Visual Basic is a single symbol. However, when the iterator method is translated to CLR metadata, it is a type and multiple methods.

Semantic model

A semantic model represents all the semantic information for a single source file. You can use it to discover the following:

- The symbols referenced at a specific location in source.
- The resultant type of any expression.
- All diagnostics, which are errors and warnings.
- How variables flow in and out of regions of source.
- The answers to more speculative questions.

Work with a workspace

3/6/2021 • 2 minutes to read • [Edit Online](#)

The **Workspaces** layer is the starting point for doing code analysis and refactoring over entire solutions. Within this layer, the Workspace API assists you in organizing all the information about the projects in a solution into a single object model, offering you direct access to compiler layer object models like source text, syntax trees, semantic models, and compilations without needing to parse files, configure options, or manage inter-project dependencies.

Host environments, like an IDE, provide a workspace for you corresponding to the open solution. It is also possible to use this model outside of an IDE by simply loading a solution file.

Workspace

A workspace is an active representation of your solution as a collection of projects, each with a collection of documents. A workspace is typically tied to a host environment that is constantly changing as a user types or manipulates properties.

The [Workspace](#) provides access to the current model of the solution. When a change in the host environment occurs, the workspace fires corresponding events, and the [Workspace.CurrentSolution](#) property is updated. For example, when the user types in a text editor corresponding to one of the source documents, the workspace uses an event to signal that the overall model of the solution has changed and which document was modified. You can then react to those changes by analyzing the new model for correctness, highlighting areas of significance, or making a suggestion for a code change.

You can also create stand-alone workspaces that are disconnected from the host environment or used in an application that has no host environment.

Solutions, projects, and documents

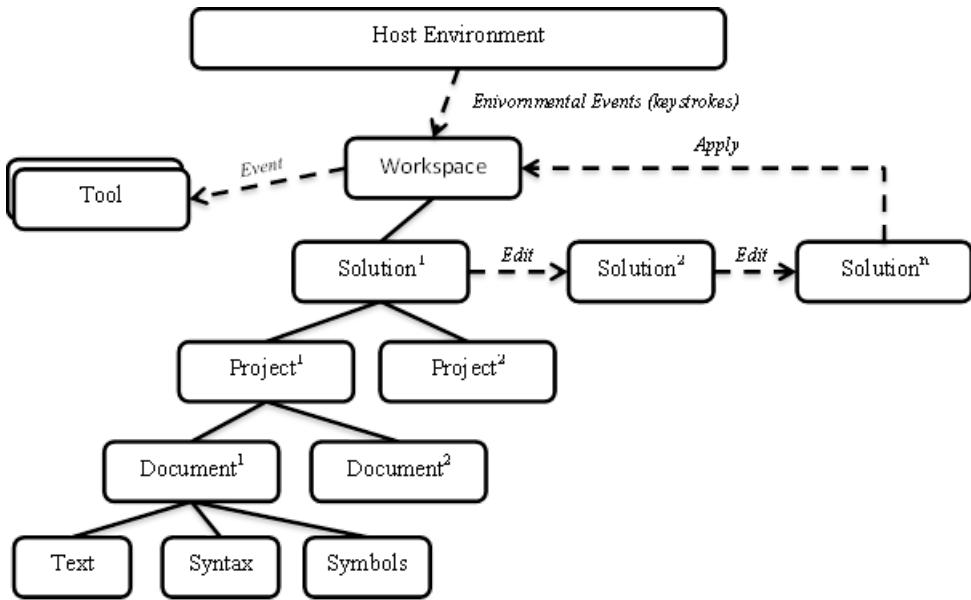
Although a workspace may change every time a key is pressed, you can work with the model of the solution in isolation.

A solution is an immutable model of the projects and documents. This means that the model can be shared without locking or duplication. After you obtain a solution instance from the [Workspace.CurrentSolution](#) property, that instance will never change. However, like with syntax trees and compilations, you can modify solutions by constructing new instances based on existing solutions and specific changes. To get the workspace to reflect your changes, you must explicitly apply the changed solution back to the workspace.

A project is a part of the overall immutable solution model. It represents all the source code documents, parse and compilation options, and both assembly and project-to-project references. From a project, you can access the corresponding compilation without needing to determine project dependencies or parse any source files.

A document is also a part of the overall immutable solution model. A document represents a single source file from which you can access the text of the file, the syntax tree, and the semantic model.

The following diagram is a representation of how the Workspace relates to the host environment, tools, and how edits are made.



Summary

Roslyn exposes a set of compiler APIs and Workspaces APIs that provides rich information about your source code and that has full fidelity with the C# and Visual Basic languages. The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code-focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and Visual Basic languages, and embedding of C# and Visual Basic in domain-specific languages.

Explore code with the Roslyn syntax visualizer in Visual Studio

3/10/2021 • 9 minutes to read • [Edit Online](#)

This article provides an overview of the Syntax Visualizer tool that ships as part of the .NET Compiler Platform ("Roslyn") SDK. The Syntax Visualizer is a tool window that helps you inspect and explore syntax trees. It's an essential tool to understand the models for code you want to analyze. It's also a debugging aid when you develop your own applications using the .NET Compiler Platform ("Roslyn") SDK. Open this tool as you create your first analyzers. The visualizer helps you understand the models used by the APIs. You can also use tools like [Sharplab](#) or [LINQPad](#) to inspect code and understand syntax trees.

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

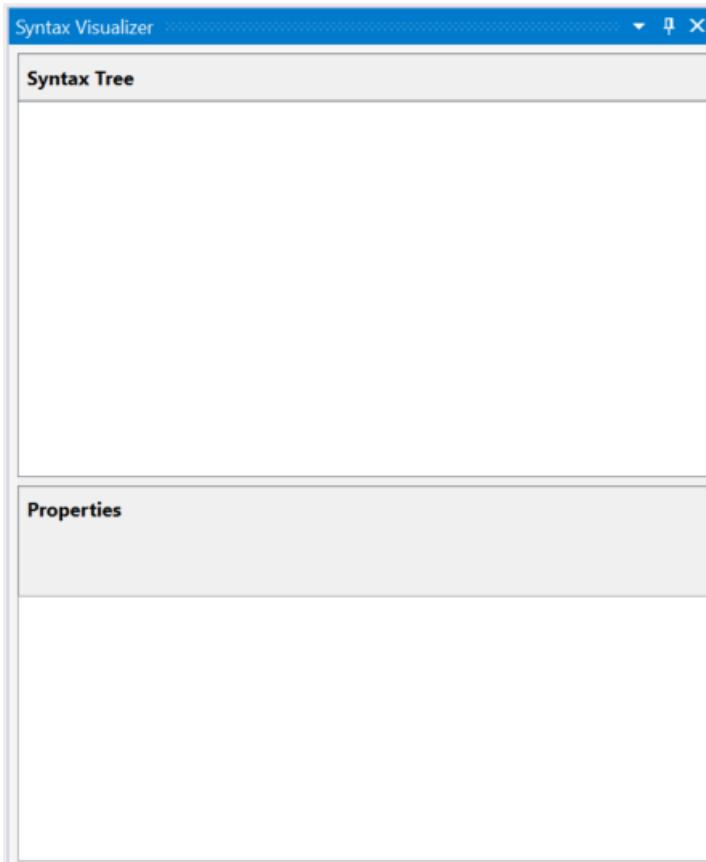
1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Familiarize yourself with the concepts used in the .NET Compiler Platform SDK by reading the [overview](#) article. It provides an introduction to syntax trees, nodes, tokens, and trivia.

Syntax Visualizer

The **Syntax Visualizer** enables inspection of the syntax tree for the C# or Visual Basic code file in the current active editor window inside the Visual Studio IDE. The visualizer can be launched by clicking on **View > Other Windows > Syntax Visualizer**. You can also use the **Quick Launch** toolbar in the upper right corner. Type "syntax", and the command to open the **Syntax Visualizer** should appear.

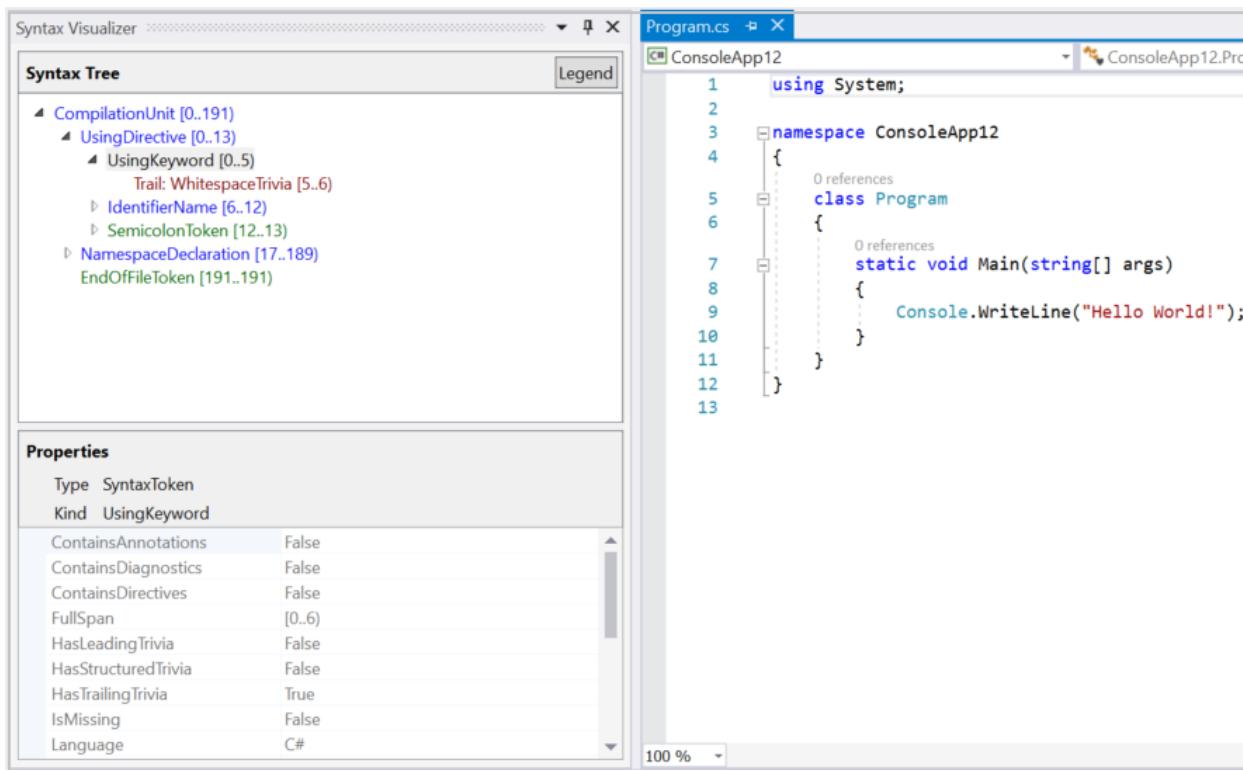
This command opens the Syntax Visualizer as a floating tool window. If you don't have a code editor window open, the display is blank, as shown in the following figure.



Dock this tool window at a convenient location inside Visual Studio, such as the left side. The Visualizer shows information about the current code file.

Create a new project using the **File > New Project** command. You can create either a Visual Basic or C# project. When Visual Studio opens the main code file for this project, the visualizer displays the syntax tree for it. You can open any existing C# / Visual Basic file in this Visual Studio instance, and the visualizer displays that file's syntax tree. If you have multiple code files open inside Visual Studio, the visualizer displays the syntax tree for the currently active code file, (the code file that has keyboard focus.)

- [C#](#)
- [Visual Basic](#)



As shown in the preceding images, the visualizer tool window displays the syntax tree at the top and a property grid at the bottom. The property grid displays the properties of the item that is currently selected in the tree, including the .NET *Type* and the *Kind*(*SyntaxKind*) of the item.

Syntax trees comprise three types of items – *nodes*, *tokens*, and *trivia*. You can read more about these types in the [Work with syntax](#) article. Items of each type are represented using a different color. Click on the ‘Legend’ button for an overview of the colors used.

Each item in the tree also displays its own **span**. The **span** is the indices (the starting and ending position) of that node in the text file. In the preceding C# example, the selected “UsingKeyword [0..5]” token has a **Span** that is five characters wide, [0..5]. The “[..]” notation means that the starting index is part of the span, but the ending index is not.

There are two ways to navigate the tree:

- Expand or click on items in the tree. The visualizer automatically selects the text corresponding to this item’s span in the code editor.
- Click or select text in the code editor. In the preceding Visual Basic example, if you select the line containing “Module Module1” in the code editor, the visualizer automatically navigates to the corresponding ModuleStatement node in the tree.

The visualizer highlights the item in the tree whose span best matches the span of the text selected in the editor.

The visualizer refreshes the tree to match modifications in the active code file. Add a call to `Console.WriteLine()` inside `Main()`. As you type, the visualizer refreshes the tree.

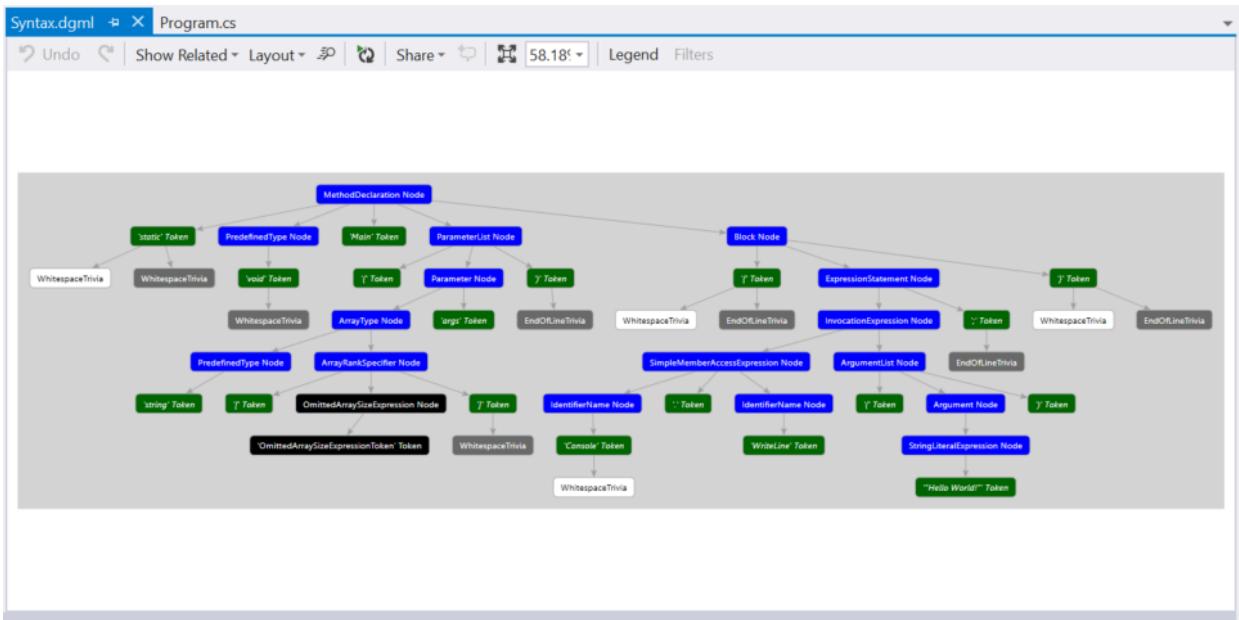
Pause typing once you have typed `Console.`. The tree has some items colored in pink. At this point, there are errors (also referred to as ‘Diagnostics’) in the typed code. These errors are attached to nodes, tokens, and trivia in the syntax tree. The visualizer shows you which items have errors attached to them highlighting the background in pink. You can inspect the errors on any item colored pink by hovering over the item. The visualizer only displays syntactic errors (those errors related to the syntax of the typed code); it doesn’t display any semantic errors.

Syntax Graphs

Right click on any item in the tree and click on **View Directed Syntax Graph**.

- C#
 - Visual Basic

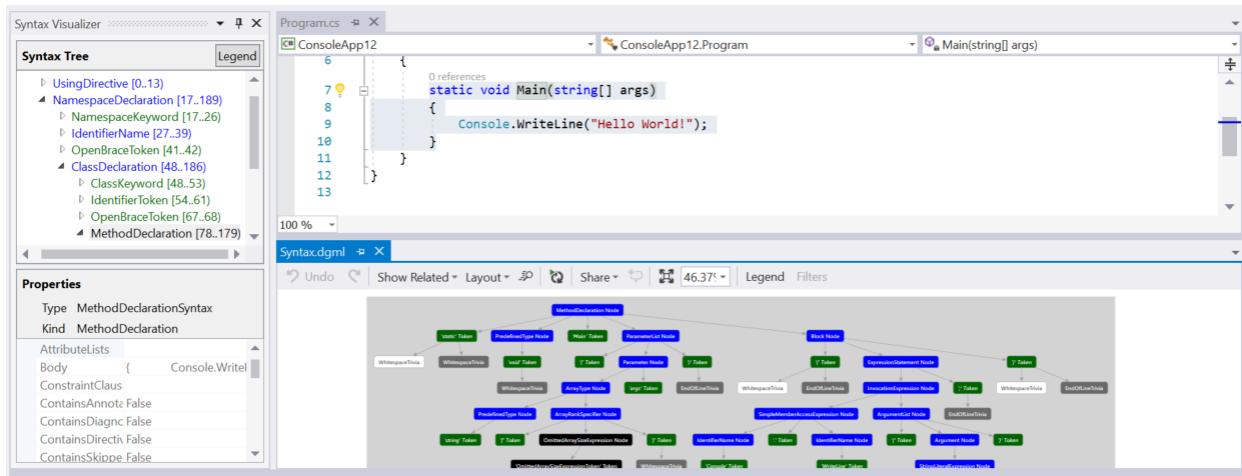
The visualizer displays a graphical representation of the subtree rooted at the selected item. Try these steps for the **MethodDeclaration** node corresponding to the `Main()` method in the C# example. The visualizer displays a syntax graph that looks as follows:



The syntax graph viewer has an option to display a legend for its coloring scheme. You can also hover over individual items in the syntax graph with the mouse to view the properties corresponding to that item.

You can view syntax graphs for different items in the tree repeatedly and the graphs will always be displayed in the same window inside Visual Studio. You can dock this window at a convenient location inside Visual Studio so that you don't have to switch between tabs to view a new syntax graph. The bottom, below code editor windows, is often convenient.

Here is the docking layout to use with the visualizer tool window and the syntax graph window:



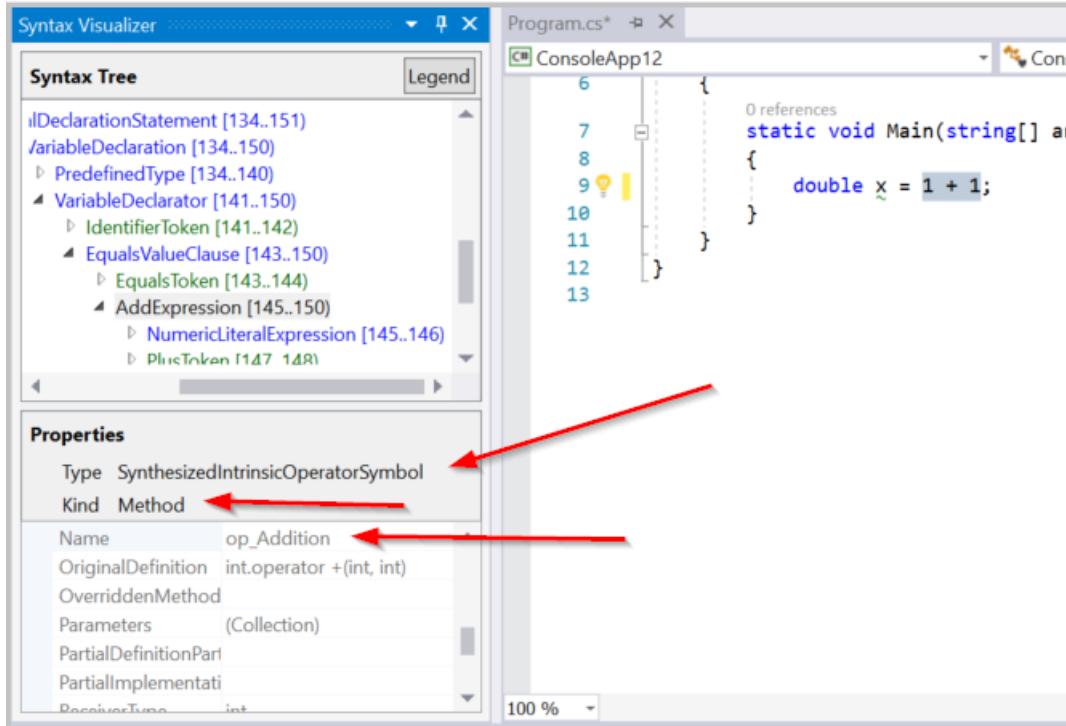
Another option is to put the syntax graph window on a second monitor, in a dual monitor setup.

Inspecting semantics

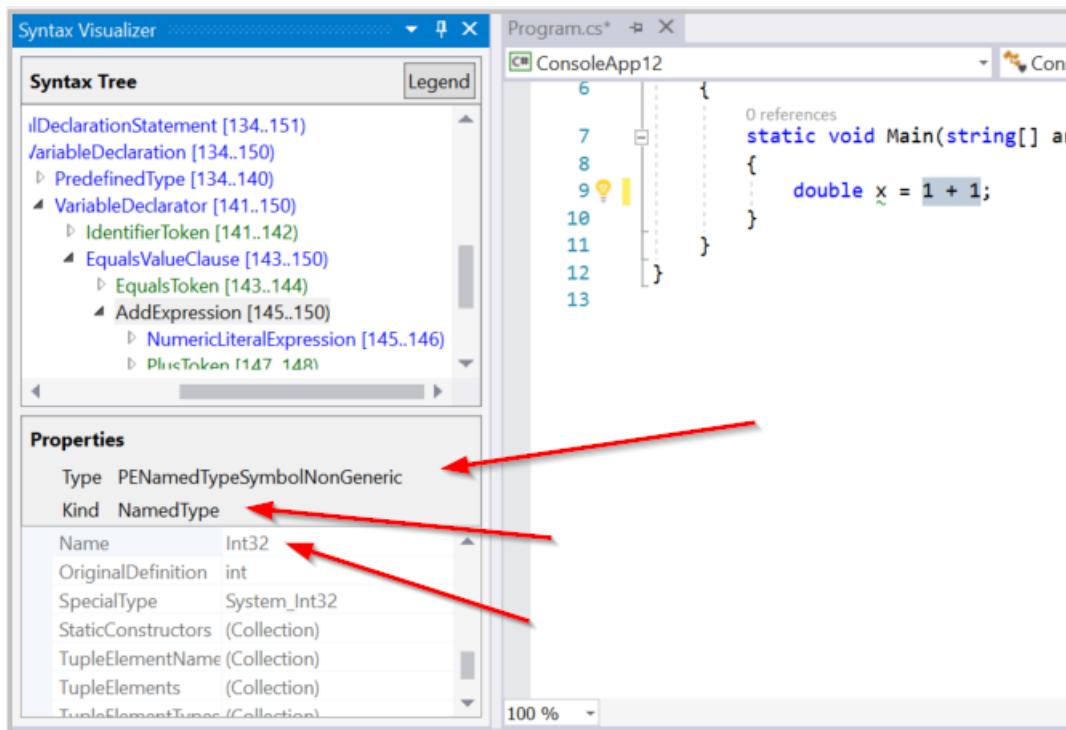
The Syntax Visualizer enables rudimentary inspection of symbols and semantic information. Type `double x = 1 + 1;` inside Main() in the C# example. Then, select the expression `1 + 1` in the code editor.

window. The visualizer highlights the **AddExpression** node in the visualizer. Right click on this **AddExpression** and click on **View Symbol (if any)**. Notice that most of the menu items have the "if any" qualifier. The Syntax Visualizer inspects properties of a Node, including properties that may not be present for all nodes.

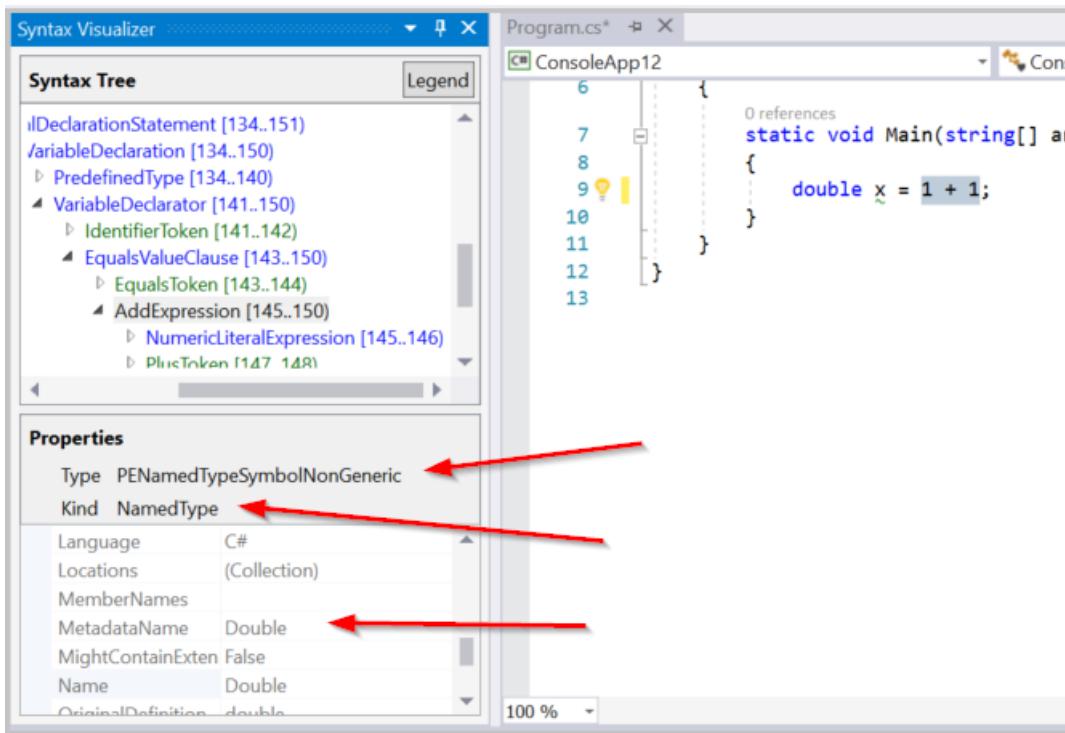
The property grid in the visualizer updates as shown in the following figure: The symbol for the expression is a **SynthesizedIntrinsicOperatorSymbol** with **Kind = Method**.



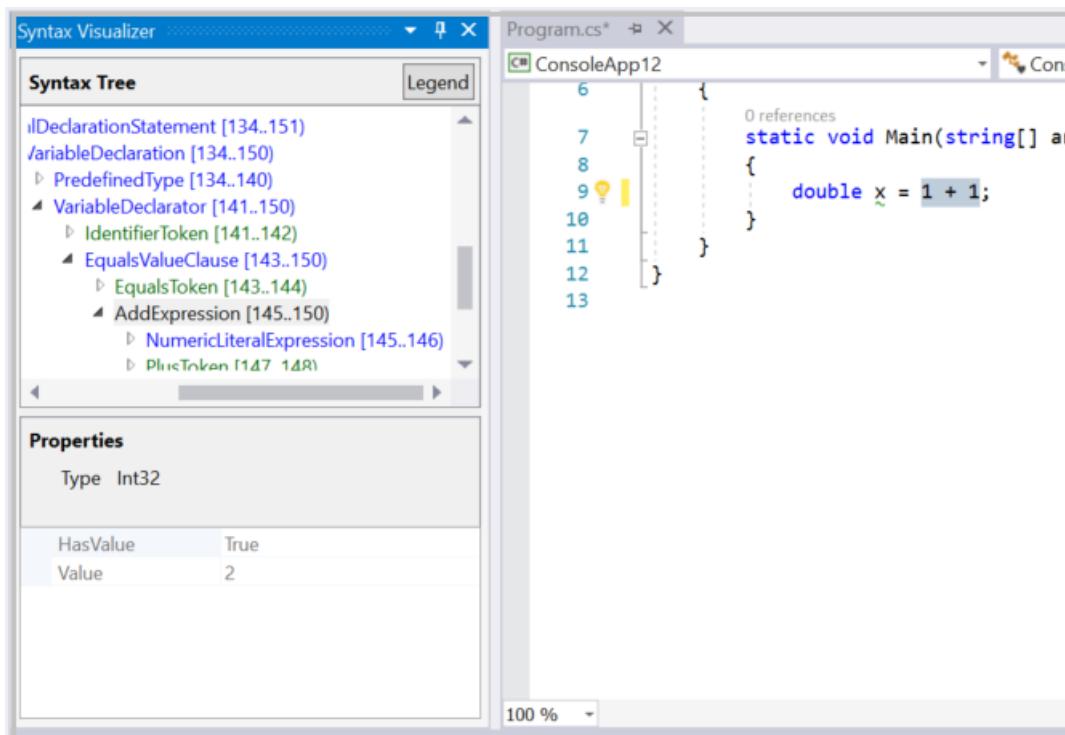
Try **View TypeSymbol (if any)** for the same **AddExpression** node. The property grid in the visualizer updates as shown in the following figure, indicating that the type of the selected expression is **Int32**.



Try **View Converted TypeSymbol (if any)** for the same **AddExpression** node. The property grid updates indicating that although the type of the expression is **Int32**, the converted type of the expression is **Double** as shown in the following figure. This node includes converted type symbol information because the **Int32** expression occurs in a context where it must be converted to a **Double**. This conversion satisfies the **Double** type specified for the variable **x** on the left-hand side of the assignment operator.



Finally, try **View Constant Value (if any)** for the same **AddExpression** node. The property grid shows that the value of the expression is a compile time constant with value `2`.



The preceding example can also be replicated in Visual Basic. Type `Dim x As Double = 1 + 1` in a Visual Basic file. Select the expression `1 + 1` in the code editor window. The visualizer highlights the corresponding **AddExpression** node in the visualizer. Repeat the preceding steps for this **AddExpression** and you should see identical results.

Examine more code in Visual Basic. Update your main Visual Basic file with the following code:

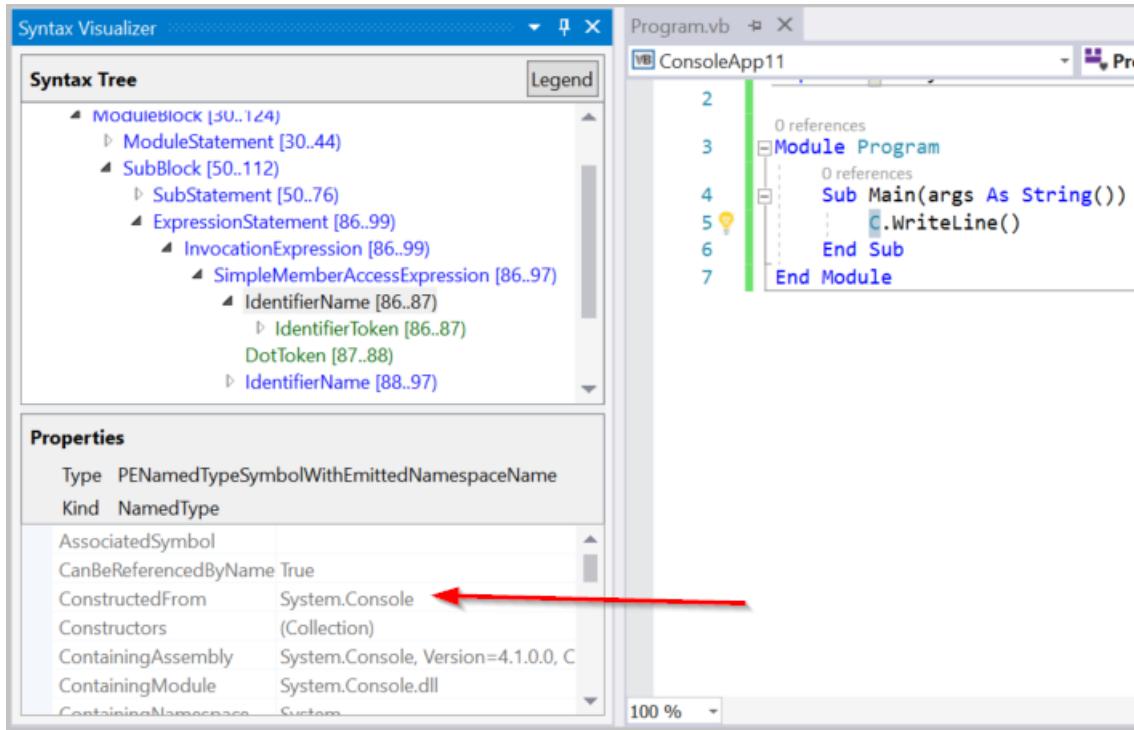
```

Imports C = System.Console

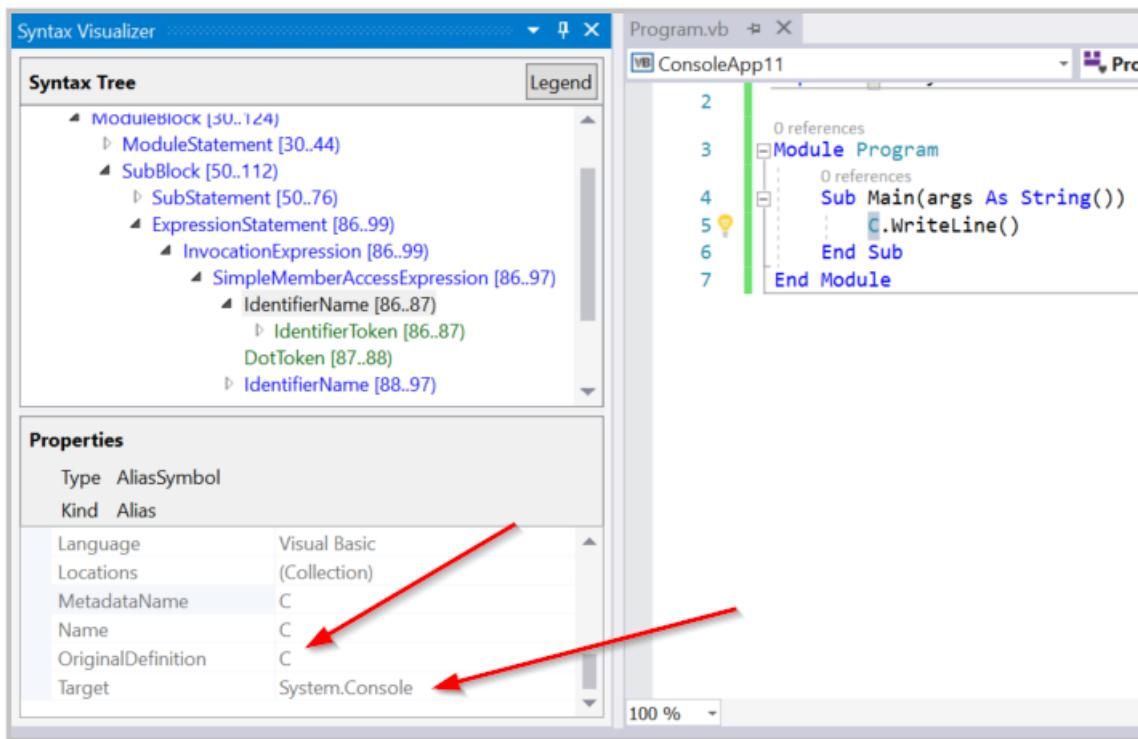
Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module

```

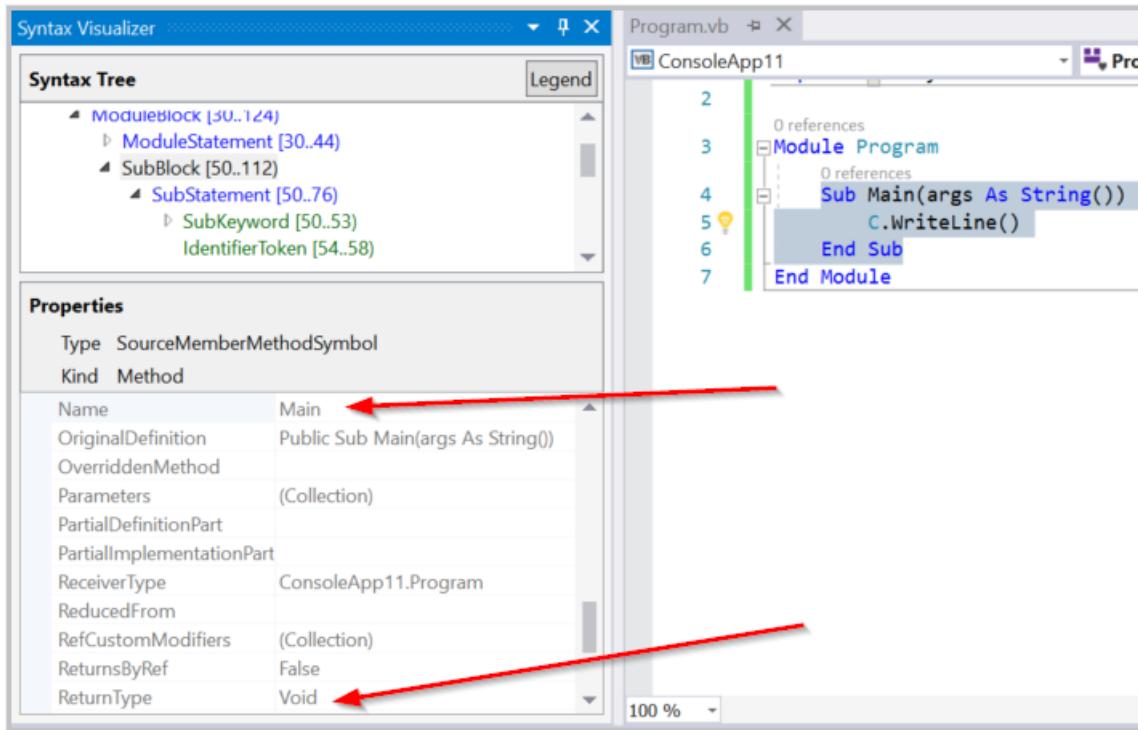
This code introduces an alias named `c` that maps to the type `System.Console` at the top of the file and uses this alias inside `Main()`. Select the use of this alias, the `c` in `C.WriteLine()`, inside the `Main()` method. The visualizer selects the corresponding **IdentifierName** node in the visualizer. Right-click this node and click on **View Symbol (if any)**. The property grid indicates that this identifier is bound to the type `System.Console` as shown in the following figure:



Try **View AliasSymbol (if any)** for the same **IdentifierName** node. The property grid indicates the identifier is an alias with name `c` that is bound to the `System.Console` target. In other words, the property grid provides information regarding the **AliasSymbol** corresponding to the identifier `c`.



Inspect the symbol corresponding to any declared type, method, property. Select the corresponding node in the visualizer and click on **View Symbol** (if any). Select the method `Sub Main()`, including the body of the method. Click on **View Symbol** (if any) for the corresponding **SubBlock** node in the visualizer. The property grid shows the **MethodSymbol** for this **SubBlock** has name `Main` with return type `Void`.



The above Visual Basic examples can be easily replicated in C#. Type `using C = System.Console;` in place of `Imports C = System.Console` for the alias. The preceding steps in C# yield identical results in the visualizer window.

Semantic inspection operations are only available on nodes. They are not available on tokens or trivia. Not all nodes have interesting semantic information to inspect. When a node doesn't have interesting semantic information, clicking on **View * Symbol** (if any) shows a blank property grid.

You can read more about APIs for performing semantic analysis in the [Work with semantics](#) overview document.

Closing the syntax visualizer

You can close the visualizer window when you are not using it to examine source code. The syntax visualizer updates its display as you navigate through code, editing and changing the source. It can get distracting when you are not using it.

Get started with syntax analysis

3/23/2021 • 13 minutes to read • [Edit Online](#)

In this tutorial, you'll explore the **Syntax API**. The Syntax API provides access to the data structures that describe a C# or Visual Basic program. These data structures have enough detail that they can fully represent any program of any size. These structures can describe complete programs that compile and run correctly. They can also describe incomplete programs, as you write them, in the editor.

To enable this rich expression, the data structures and APIs that make up the Syntax API are necessarily complex. Let's start with what the data structure looks like for the typical "Hello World" program:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Look at the text of the previous program. You recognize familiar elements. The entire text represents a single source file, or a **compilation unit**. The first three lines of that source file are **using directives**. The remaining source is contained in a **namespace declaration**. The namespace declaration contains a child **class declaration**. The class declaration contains one **method declaration**.

The Syntax API creates a tree structure with the root representing the compilation unit. Nodes in the tree represent the using directives, namespace declaration and all the other elements of the program. The tree structure continues down to the lowest levels: the string "Hello World!" is a **string literal token** that is a descendent of an **argument**. The Syntax API provides access to the structure of the program. You can query for specific code practices, walk the entire tree to understand the code, and create new trees by modifying the existing tree.

That brief description provides an overview of the kind of information accessible using the Syntax API. The Syntax API is nothing more than a formal API that describes the familiar code constructs you know from C#. The full capabilities include information about how the code is formatted including line breaks, white space, and indenting. Using this information, you can fully represent the code as written and read by human programmers or the compiler. Using this structure enables you to interact with the source code on a deeply meaningful level. It's no longer text strings, but data that represents the structure of a C# program.

To get started, you'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension

development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Understanding syntax trees

You use the Syntax API for any analysis of the structure of C# code. The **Syntax API** exposes the parsers, the syntax trees, and utilities for analyzing and constructing syntax trees. It's how you search code for specific syntax elements or read the code for a program.

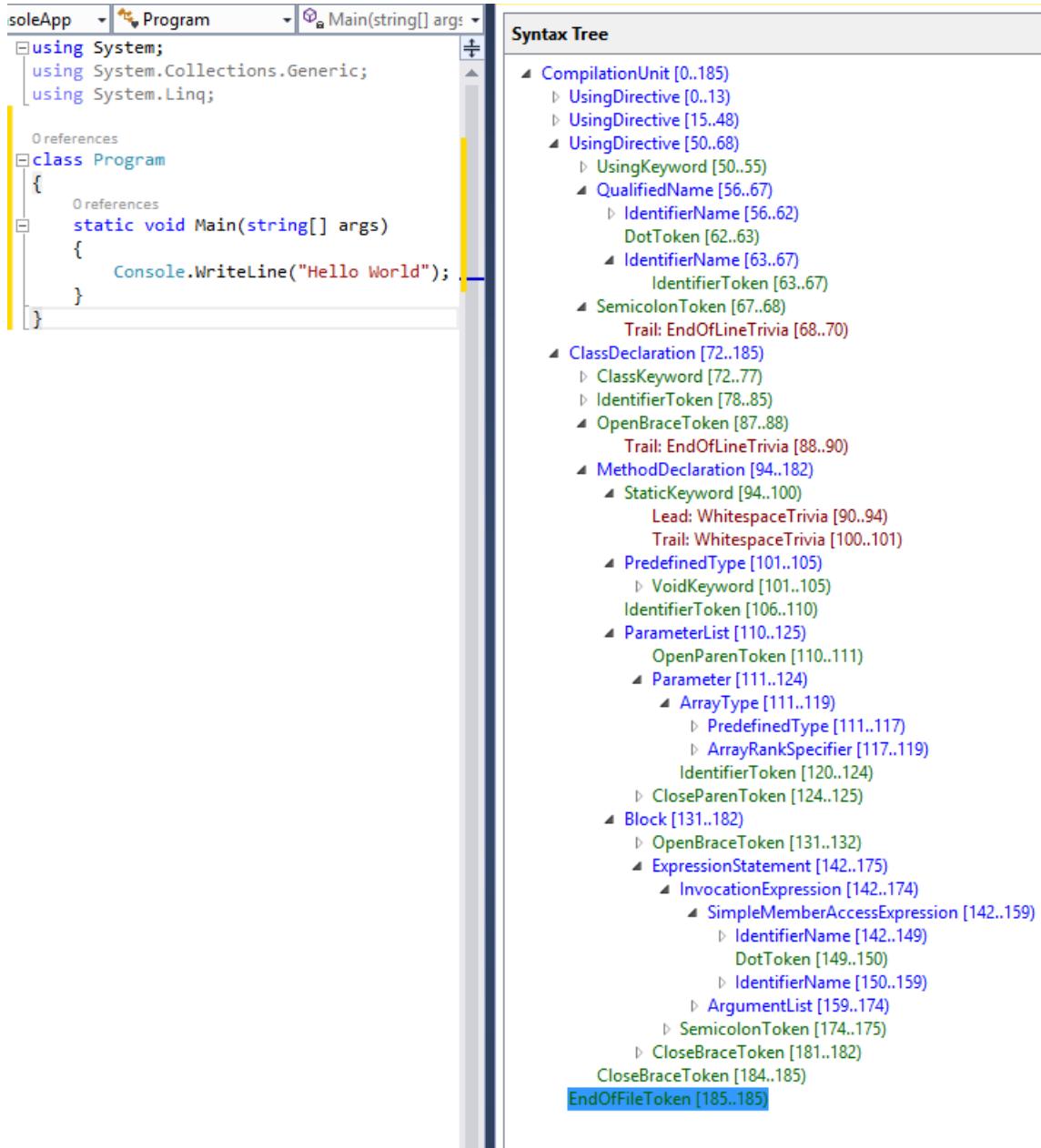
A syntax tree is a data structure used by the C# and Visual Basic compilers to understand C# and Visual Basic programs. Syntax trees are produced by the same parser that runs when a project is built or a developer hits F5. The syntax trees have full-fidelity with the language; every bit of information in a code file is represented in the tree. Writing a syntax tree to text reproduces the exact original text that was parsed. The syntax trees are also **immutable**; once created a syntax tree can never be changed. Consumers of the trees can analyze the trees on multiple threads, without locks or other concurrency measures, knowing the data never changes. You can use APIs to create new trees that are the result of modifying an existing tree.

The four primary building blocks of syntax trees are:

- The [Microsoft.CodeAnalysis.SyntaxTree](#) class, an instance of which represents an entire parse tree. [SyntaxTree](#) is an abstract class that has language-specific derivatives. You use the parse methods of the [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree](#) (or [Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree](#)) class to parse text in C# (or Visual Basic).
- The [Microsoft.CodeAnalysis.SyntaxNode](#) class, instances of which represent syntactic constructs such as declarations, statements, clauses, and expressions.
- The [Microsoft.CodeAnalysis.SyntaxToken](#) structure, which represents an individual keyword, identifier, operator, or punctuation.
- And lastly the [Microsoft.CodeAnalysis.SyntaxTrivia](#) structure, which represents syntactically insignificant bits of information such as the white space between tokens, preprocessing directives, and comments.

Trivia, tokens, and nodes are composed hierarchically to form a tree that completely represents everything in a fragment of Visual Basic or C# code. You can see this structure using the **Syntax Visualizer** window. In Visual Studio, choose **View > Other Windows > Syntax Visualizer**. For example, the preceding C# source file examined using the **Syntax Visualizer** looks like the following figure:

SyntaxNode: Blue | SyntaxToken: Green | SyntaxTrivia: Red



By navigating this tree structure, you can find any statement, expression, token, or bit of white space in a code file.

While you can find anything in a code file using the Syntax APIs, most scenarios involve examining small snippets of code, or searching for particular statements or fragments. The two examples that follow show typical uses to browse the structure of code, or search for single statements.

Traversing trees

You can examine the nodes in a syntax tree in two ways. You can traverse the tree to examine each node, or you can query for specific elements or nodes.

Manual traversal

You can see the finished code for this sample in [our GitHub repository](#).

NOTE

The Syntax Tree types use inheritance to describe the different syntax elements that are valid at different locations in the program. Using these APIs often means casting properties or collection members to specific derived types. In the following examples, the assignment and the casts are separate statements, using explicitly typed variables. You can read the code to see the return types of the API and the runtime type of the objects returned. In practice, it's more common to use implicitly typed variables and rely on API names to describe the type of objects being examined.

Create a new C# Stand-Alone Code Analysis Tool project:

- In Visual Studio, choose **File > New > Project** to display the New Project dialog.
- Under **Visual C# > Extensibility**, choose **Stand-Alone Code Analysis Tool**.
- Name your project "**SyntaxTreeManualTraversal**" and click OK.

You're going to analyze the basic "Hello World!" program shown earlier. Add the text for the Hello World program as a constant in your `Program` class:

```
const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}";
```

Next, add the following code to build the **syntax tree** for the code text in the `programText` constant. Add the following line to your `Main` method:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Those two lines create the tree and retrieve the root node of that tree. You can now examine the nodes in the tree. Add these lines to your `Main` method to display some of the properties of the root node in the tree:

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"{element.Name}");
```

Run the application to see what your code has discovered about the root node in this tree.

Typically, you'd traverse the tree to learn about the code. In this example, you're analyzing code you know to explore the APIs. Add the following code to examine the first member of the `root` node:

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

That member is a [Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax](#). It represents everything in the scope of the `namespace HelloWorld` declaration. Add the following code to examine what nodes are declared inside the `HelloWorld` namespace:

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared in this namespace.");
WriteLine($"The first member is a {helloWorldDeclaration.Members[0].Kind()}.");
```

Run the program to see what you've learned.

Now that you know the declaration is a [Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax](#), declare a new variable of that type to examine the class declaration. This class only contains one member: the `Main` method. Add the following code to find the `Main` method, and cast it to a [Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax](#).

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in the
{programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

The method declaration node contains all the syntactic information about the method. Let's display the return type of the `Main` method, the number and types of the arguments, and the body text of the method. Add the following code:

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is {mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count} parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is {item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method follows:");
WriteLine(mainDeclaration.Body.ToString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Run the program to see all the information you've discovered about this program:

```
The tree is a CompilationUnit node.  
The tree has 1 elements in it.  
The tree has 4 using statements. They are:  
    System  
    System.Collections  
    System.Linq  
    System.Text  
The first member is a NamespaceDeclaration.  
There are 1 members declared in this namespace.  
The first member is a ClassDeclaration.  
There are 1 members declared in the Program class.  
The first member is a MethodDeclaration.  
The return type of the Main method is void.  
The method has 1 parameters.  
The type of the args parameter is string[].  
The body text of the Main method follows:  
{  
    Console.WriteLine("Hello, World!");  
}
```

Query methods

In addition to traversing trees, you can also explore the syntax tree using the query methods defined on [Microsoft.CodeAnalysis.SyntaxNode](#). These methods should be immediately familiar to anyone familiar with XPath. You can use these methods with LINQ to quickly find things in a tree. The [SyntaxNode](#) has query methods such as [DescendantNodes](#), [AncestorsAndSelf](#) and [ChildNodes](#).

You can use these query methods to find the argument to the `Main` method as an alternative to navigating the tree. Add the following code to the bottom of your `Main` method:

```
var firstParameters = from methodDeclaration in root.DescendantNodes()  
                      .OfType<MethodDeclarationSyntax>()  
                      where methodDeclaration.Identifier.ValueText == "Main"  
                      select methodDeclaration.ParameterList.Parameters.First();  
  
var argsParameter2 = firstParameters.Single();  
  
WriteLine(argsParameter == argsParameter2);
```

The first statement uses a LINQ expression and the [DescendantNodes](#) method to locate the same parameter as in the previous example.

Run the program, and you can see that the LINQ expression found the same parameter as manually navigating the tree.

The sample uses `WriteLine` statements to display information about the syntax trees as they are traversed. You can also learn much more by running the finished program under the debugger. You can examine more of the properties and methods that are part of the syntax tree created for the hello world program.

Syntax walkers

Often you want to find all nodes of a specific type in a syntax tree, for example, every property declaration in a file. By extending the [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker](#) class and overriding the [VisitPropertyDeclaration\(PropertyDeclarationSyntax\)](#) method, you process every property declaration in a syntax tree without knowing its structure beforehand. [CSharpSyntaxWalker](#) is a specific kind of [CSharpSyntaxVisitor](#) that recursively visits a node and each of its children.

This example implements a [CSharpSyntaxWalker](#) that examines a syntax tree. It collects `using` directives it finds that aren't importing a `System` namespace.

Create a new C# Stand-Alone Code Analysis Tool project; name it "SyntaxWalker."

You can see the finished code for this sample in [our GitHub repository](#). The sample on GitHub contains both projects described in this tutorial.

As in the previous sample, you can define a string constant to hold the text of the program you're going to analyze:

```
const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

This source text contains `using` directives scattered across four different locations: the file-level, in the top-level namespace, and in the two nested namespaces. This example highlights a core scenario for using the [CSharpSyntaxWalker](#) class to query code. It would be cumbersome to visit every node in the root syntax tree to find `using` declarations. Instead, you create a derived class and override the method that gets called only when the current node in the tree is a `using` directive. Your visitor does not do any work on any other node types. This single method examines each of the `using` statements and builds a collection of the namespaces that aren't in the `System` namespace. You build a [CSharpSyntaxWalker](#) that examines all the `using` statements, but only the `using` statements.

Now that you've defined the program text, you need to create a `SyntaxTree` and get the root of that tree:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Next, create a new class. In Visual Studio, choose **Project > Add New Item**. In the **Add New Item** dialog type `UsingCollector.cs` as the filename.

You implement the `using` visitor functionality in the `UsingCollector` class. Start by making the `UsingCollector` class derive from [CSharpSyntaxWalker](#).

```
class UsingCollector : CSharpSyntaxWalker
```

You need storage to hold the namespace nodes that you're collecting. Declare a public read-only property in the `UsingCollector` class; you use this variable to store the `UsingDirectiveSyntax` nodes you find:

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new List<UsingDirectiveSyntax>();
```

The base class, `CSharpSyntaxWalker` implements the logic to visit each node in the syntax tree. The derived class overrides the methods called for the specific nodes you're interested in. In this case, you're interested in any `using` directive. That means you must override the `VisitUsingDirective(UsingDirectiveSyntax)` method. The one argument to this method is a `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` object. That's an important advantage to using the visitors: they call the overridden methods with arguments already cast to the specific node type. The `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` class has a `Name` property that stores the name of the namespace being imported. It is a `Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax`. Add the following code in the `VisitUsingDirective(UsingDirectiveSyntax)` override:

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\\t\\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

As with the earlier example, you've added a variety of `WriteLine` statements to aid in understanding of this method. You can see when it's called, and what arguments are passed to it each time.

Finally, you need to add two lines of code to create the `usingCollector` and have it visit the root node, collecting all the `using` statements. Then, add a `foreach` loop to display all the `using` statements your collector found:

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compile and run the program. You should see the following output:

```
VisitUsingDirective called with System.  
VisitUsingDirective called with System.Collections.Generic.  
VisitUsingDirective called with System.Linq.  
VisitUsingDirective called with System.Text.  
VisitUsingDirective called with Microsoft.CodeAnalysis.  
    Success. Adding Microsoft.CodeAnalysis.  
VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.  
    Success. Adding Microsoft.CodeAnalysis.CSharp.  
VisitUsingDirective called with Microsoft.  
    Success. Adding Microsoft.  
VisitUsingDirective called with System.ComponentModel.  
VisitUsingDirective called with Microsoft.Win32.  
    Success. Adding Microsoft.Win32.  
VisitUsingDirective called with System.Runtime.InteropServices.  
VisitUsingDirective called with System.CodeDom.  
VisitUsingDirective called with Microsoft.CSharp.  
    Success. Adding Microsoft.CSharp.  
Microsoft.CodeAnalysis  
Microsoft.CodeAnalysis.CSharp  
Microsoft  
Microsoft.Win32  
Microsoft.CSharp  
Press any key to continue . . .
```

Congratulations! You've used the **Syntax API** to locate specific kinds of C# statements and declarations in C# source code.

Get started with semantic analysis

3/23/2021 • 8 minutes to read • [Edit Online](#)

This tutorial assumes you're familiar with the Syntax API. The [get started with syntax analysis](#) article provides sufficient introduction.

In this tutorial, you explore the **Symbol** and **Binding APIs**. These APIs provide information about the *semantic meaning* of a program. They enable you to ask and answer questions about the types represented by any symbol in your program.

You'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The **.NET Compiler Platform SDK** is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Understanding Compilations and Symbols

As you work more with the **.NET Compiler SDK**, you become familiar with the distinctions between Syntax API and the Semantic API. The **Syntax API** allows you to look at the *structure* of a program. However, often you want richer information about the semantics or *meaning* of a program. While a loose code file or snippet of Visual Basic or C# code can be syntactically analyzed in isolation, it's not meaningful to ask questions such as "what's the type of this variable" in a vacuum. The meaning of a type name may be dependent on assembly references, namespace imports, or other code files. Those questions are answered using the **Semantic API**, specifically the [Microsoft.CodeAnalysis.Compilation](#) class.

An instance of [Compilation](#) is analogous to a single project as seen by the compiler and represents everything needed to compile a Visual Basic or C# program. The **compilation** includes the set of source files to be compiled, assembly references, and compiler options. You can reason about the meaning of the code using all the other information in this context. A [Compilation](#) allows you to find **Symbols** - entities such as types, namespaces, members, and variables which names and other expressions refer to. The process of associating names and expressions with **Symbols** is called **Binding**.

Like [Microsoft.CodeAnalysis.SyntaxTree](#), [Compilation](#) is an abstract class with language-specific derivatives. When creating an instance of [Compilation](#), you must invoke a factory method on the [Microsoft.CodeAnalysis.CSharp.CSharpCompilation](#) (or [Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation](#)) class.

Querying symbols

In this tutorial, you look at the "Hello World" program again. This time, you query the symbols in the program to understand what types those symbols represent. You query for the types in a namespace, and learn to find the methods available on a type.

You can see the finished code for this sample in [our GitHub repository](#).

NOTE

The Syntax Tree types use inheritance to describe the different syntax elements that are valid at different locations in the program. Using these APIs often means casting properties or collection members to specific derived types. In the following examples, the assignment and the casts are separate statements, using explicitly typed variables. You can read the code to see the return types of the API and the runtime type of the objects returned. In practice, it's more common to use implicitly typed variables and rely on API names to describe the type of objects being examined.

Create a new C# Stand-Alone Code Analysis Tool project:

- In Visual Studio, choose **File > New > Project** to display the New Project dialog.
- Under **Visual C# > Extensibility**, choose **Stand-Alone Code Analysis Tool**.
- Name your project "**SemanticQuickStart**" and click OK.

You're going to analyze the basic "Hello World!" program shown earlier. Add the text for the Hello World program as a constant in your `Program` class:

```
const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}";
```

Next, add the following code to build the syntax tree for the code text in the `programText` constant. Add the following line to your `Main` method:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Next, build a [CSharpCompilation](#) from the tree you already created. The "Hello World" sample relies on the [String](#) and [Console](#) types. You need to reference the assembly that declares those two types in your compilation. Add the following line to your `Main` method to create a compilation of your syntax tree, including the reference to the appropriate assembly:

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

The [CSharpCompilation.AddReferences](#) method adds references to the compilation. The [MetadataReference.CreateFromFile](#) method loads an assembly as a reference.

Querying the semantic model

Once you have a [Compilation](#) you can ask it for a [SemanticModel](#) for any [SyntaxTree](#) contained in that [Compilation](#). You can think of the semantic model as the source for all the information you would normally get from intellisense. A [SemanticModel](#) can answer questions like "What names are in scope at this location?", "What members are accessible from this method?", "What variables are used in this block of text?", and "What does this name/expression refer to?" Add this statement to create the semantic model:

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Binding a name

The [Compilation](#) creates the [SemanticModel](#) from the [SyntaxTree](#). After creating the model, you can query it to find the first `using` directive, and retrieve the symbol information for the `System` namespace. Add these two lines to your `Main` method to create the semantic model and retrieve the symbol for the first using statement:

```
// Use the syntax tree to find "using System;"  
UsingDirectiveSyntax usingSystem = root.Usings[0];  
NameSyntax systemName = usingSystem.Name;  
  
// Use the semantic model for symbol information:  
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

The preceding code shows how to bind the name in the first `using` directive to retrieve a [Microsoft.CodeAnalysis.SymbolInfo](#) for the `System` namespace. The preceding code also illustrates that you use the **syntax model** to find the structure of the code; you use the **semantic model** to understand its meaning. The **syntax model** finds the string `System` in the using statement. The **semantic model** has all the information about the types defined in the `System` namespace.

From the [SymbolInfo](#) object you can obtain the [Microsoft.CodeAnalysis.ISymbol](#) using the [SymbolInfo.Symbol](#) property. This property returns the symbol this expression refers to. For expressions that don't refer to anything (such as numeric literals) this property is `null`. When the [SymbolInfo.Symbol](#) is not null, the [ISymbol.Kind](#) denotes the type of the symbol. In this example, the [ISymbol.Kind](#) property is a [SymbolKind.Namespace](#). Add the following code to your `Main` method. It retrieves the symbol for the `System` namespace and then displays all the child namespaces declared in the `System` namespace:

```

var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}

```

Run the program and you should see the following output:

```

System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .

```

NOTE

The output does not include every namespace that is a child namespace of the `System` namespace. It displays every namespace that is present in this compilation, which only references the assembly where `System.String` is declared. Any namespaces declared in other assemblies are not known to this compilation.

Binding an expression

The preceding code shows how to find a symbol by binding to a name. There are other expressions in a C# program that can be bound that aren't names. To demonstrate this capability, let's access the binding to a simple string literal.

The "Hello World" program contains a [Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax](#), the "Hello, World!" string displayed to the console.

You find the "Hello, World!" string by locating the single string literal in the program. Then, once you've located the syntax node, get the type info for that node from the semantic model. Add the following code to your `Main` method:

```

// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
    .OfType<LiteralExpressionSyntax>()
    .Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);

```

The [Microsoft.CodeAnalysis.TypeInfo](#) struct includes a `TypeInfo.Type` property that enables access to the semantic information about the type of the literal. In this example, that's the `string` type. Add a declaration that assigns this property to a local variable:

```

var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;

```

To finish this tutorial, let's build a LINQ query that creates a sequence of all the public methods declared on the `string` type that return a `string`. This query gets complex, so let's build it line by line, then reconstruct it as a single query. The source for this query is the sequence of all members declared on the `string` type:

```
var allMembers = stringTypeSymbol.GetMembers();
```

That source sequence contains all members, including properties and fields, so filter it using the `ImmutableArray<T>.OfType` method to find elements that are `Microsoft.CodeAnalysis.IMethodSymbol` objects:

```
var methods = allMembers.OfType<IMethodSymbol>();
```

Next, add another filter to return only those methods that are public and return a `string`:

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Select only the name property, and only distinct names by removing any overloads:

```
var distinctMethods = publicStringReturningMethods.Select(m => m.Name).Distinct();
```

You can also build the full query using the LINQ query syntax, and then display all the method names in the console:

```
foreach (string name in (from method in stringTypeSymbol
    .GetMembers().OfType<IMethodSymbol>()
    where method.ReturnType.Equals(stringTypeSymbol) &&
    method.DeclaredAccessibility == Accessibility.Public
    select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Build and run the program. You should see the following output:

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

You've used the Semantic API to find and display information about the symbols that are part of this program.

Get started with syntax transformation

3/31/2021 • 12 minutes to read • [Edit Online](#)

This tutorial builds on concepts and techniques explored in the [Get started with syntax analysis](#) and [Get started with semantic analysis](#) quickstarts. If you haven't already, you should complete those quickstarts before beginning this one.

In this quickstart, you explore techniques for creating and transforming syntax trees. In combination with the techniques you learned in previous quickstarts, you create your first command-line refactoring!

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The **.NET Compiler Platform SDK** is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Immutability and the .NET compiler platform

Immutability is a fundamental tenet of the .NET compiler platform. Immutable data structures can't be changed after they're created. Immutable data structures can be safely shared and analyzed by multiple consumers simultaneously. There's no danger that one consumer affects another in unpredictable ways. Your analyzer doesn't need locks or other concurrency measures. This rule applies to syntax trees, compilations, symbols, semantic models, and every other data structure you encounter. Instead of modifying existing structures, APIs create new objects based on specified differences to the old ones. You apply this concept to syntax trees to create new trees using transformations.

Create and transform trees

You choose one of two strategies for syntax transformations. **Factory methods** are best used when you're searching for specific nodes to replace, or specific locations where you want to insert new code. **Rewriters** are best when you want to scan an entire project for code patterns that you want to replace.

Create nodes with factory methods

The first syntax transformation demonstrates the factory methods. You're going to replace a

`using System.Collections;` statement with a `using System.Collections.Generic;` statement. This example demonstrates how you create `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode` objects using the `Microsoft.CodeAnalysis.CSharp.SyntaxFactory` factory methods. For each kind of `node`, `token`, or `trivia`, there's a factory method that creates an instance of that type. You create syntax trees by composing nodes hierarchically in a bottom-up fashion. Then, you'll transform the existing program by replacing existing nodes with the new tree you've created.

Start Visual Studio, and create a new **C# Stand-Alone Code Analysis Tool** project. In Visual Studio, choose **File > New > Project** to display the New Project dialog. Under **Visual C# > Extensibility** choose a **Stand-Alone Code Analysis Tool**. This quickstart has two example projects, so name the solution `SyntaxTransformationQuickStart`, and name the project `ConstructionCS`. Click **OK**.

This project uses the `Microsoft.CodeAnalysis.CSharp.SyntaxFactory` class methods to construct a `Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax` representing the `System.Collections.Generic` namespace.

Add the following using directive to the top of the `Program.cs`.

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

You'll create **name syntax nodes** to build the tree that represents the `using System.Collections.Generic;` statement. `NameSyntax` is the base class for four types of names that appear in C#. You compose these four types of names together to create any name that can appear in the C# language:

- `Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax`, which represents simple single identifier names like `System` and `Microsoft`.
- `Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax`, which represents a generic type or method name such as `List<int>`.
- `Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax`, which represents a qualified name of the form `<left-name>.<right-identifier-or-generic-name>` such as `System.IO`.
- `Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax`, which represents a name using an assembly extern alias such a `LibraryV2::Foo`.

You use the `IdentifierName(String)` method to create a `NameSyntax` node. Add the following code in your `Main` method in `Program.cs`:

```
NameSyntax name = IdentifierName("System");
WriteLine($"\\tCreated the identifier {name}");
```

The preceding code creates an `IdentifierNameSyntax` object and assigns it to the variable `name`. Many of the Roslyn APIs return base classes to make it easier to work with related types. The variable `name`, an `NameSyntax`, can be reused as you build the `QualifiedNameSyntax`. Don't use type inference as you build the sample. You'll automate that step in this project.

You've created the name. Now, it's time to build more nodes into the tree by building a `QualifiedNameSyntax`. The new tree uses `name` as the left of the name, and a new `IdentifierNameSyntax` for the `Collections`

namespace as the right side of the [QualifiedNameSyntax](#). Add the following code to `Program.cs`:

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Run the code again, and see the results. You're building a tree of nodes that represents code. You'll continue this pattern to build the [QualifiedNameSyntax](#) for the namespace `System.Collections.Generic`. Add the following code to `Program.cs`:

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Run the program again to see that you've built the tree for the code to add.

Create a modified tree

You've built a small syntax tree that contains one statement. The APIs to create new nodes are the right choice to create single statements or other small code blocks. However, to build larger blocks of code, you should use methods that replace nodes or insert nodes into an existing tree. Remember that syntax trees are immutable. The [Syntax API](#) doesn't provide any mechanism for modifying an existing syntax tree after construction. Instead, it provides methods that produce new trees based on changes to existing ones. `With*` methods are defined in concrete classes that derive from [SyntaxNode](#) or in extension methods declared in the [SyntaxNodeExtensions](#) class. These methods create a new node by applying changes to an existing node's child properties. Additionally, the [ReplaceNode](#) extension method can be used to replace a descendent node in a subtree. This method also updates the parent to point to the newly created child and repeats this process up the entire tree - a process known as *re-spinning* the tree.

The next step is to create a tree that represents an entire (small) program and then modify it. Add the following code to the beginning of the `Program` class:

```
private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}";
```

NOTE

The example code uses the `System.Collections` namespace and not the `System.Collections.Generic` namespace.

Next, add the following code to the bottom of the `Main` method to parse the text and create a tree:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

This example uses the [WithName\(NameSyntax\)](#) method to replace the name in a [UsingDirectiveSyntax](#) node with the one constructed in the preceding code.

Create a new [UsingDirectiveSyntax](#) node using the [WithName\(NameSyntax\)](#) method to update the [System.Collections](#) name with the name you created in the preceding code. Add the following code to the bottom of the [Main](#) method:

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Run the program and look carefully at the output. The [newUsing](#) hasn't been placed in the root tree. The original tree hasn't been changed.

Add the following code using the [ReplaceNode](#) extension method to create a new tree. The new tree is the result of replacing the existing import with the updated [newUsing](#) node. You assign this new tree to the existing [root](#) variable:

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Run the program again. This time the tree now correctly imports the [System.Collections.Generic](#) namespace.

Transform trees using [SyntaxRewriters](#)

The [With*](#) and [ReplaceNode](#) methods provide convenient means to transform individual branches of a syntax tree. The [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter](#) class performs multiple transformations on a syntax tree. The [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter](#) class is a subclass of [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>](#). The [CSharpSyntaxRewriter](#) applies a transformation to a specific type of [SyntaxNode](#). You can apply transformations to multiple types of [SyntaxNode](#) objects wherever they appear in a syntax tree. The second project in this quickstart creates a command-line refactoring that removes explicit types in local variable declarations anywhere that type inference could be used.

Create a new **C# Stand-Alone Code Analysis Tool** project. In Visual Studio, right-click the [SyntaxTransformationQuickStart](#) solution node. Choose **Add > New Project** to display the **New Project dialog**. Under **Visual C# > Extensibility**, choose **Stand-Alone Code Analysis Tool**. Name your project [TransformationCS](#) and click **OK**.

The first step is to create a class that derives from [CSharpSyntaxRewriter](#) to perform your transformations. Add a new class file to the project. In Visual Studio, choose **Project > Add Class....** In the **Add New Item** dialog type [TypeInferenceRewriter.cs](#) as the filename.

Add the following using directives to the [TypeInferenceRewriter.cs](#) file:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Next, make the [TypeInferenceRewriter](#) class extend the [CSharpSyntaxRewriter](#) class:

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Add the following code to declare a private read-only field to hold a [SemanticModel](#) and initialize it in the constructor. You will need this field later on to determine where type inference can be used:

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Override the [VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#) method:

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{
}
```

NOTE

Many of the Roslyn APIs declare return types that are base classes of the actual runtime types returned. In many scenarios, one kind of node may be replaced by another kind of node entirely - or even removed. In this example, the [VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#) method returns a [SyntaxNode](#), instead of the derived type of [LocalDeclarationStatementSyntax](#). This rewriter returns a new [LocalDeclarationStatementSyntax](#) node based on the existing one.

This quickstart handles local variable declarations. You could extend it to other declarations such as `foreach` loops, `for` loops, LINQ expressions, and lambda expressions. Furthermore this rewriter will only transform declarations of the simplest form:

```
Type variable = expression;
```

If you want to explore on your own, consider extending the finished sample for these types of variable declarations:

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
    variable2 = expression2;
// No initializer.
Type variable;
```

Add the following code to the body of the [VisitLocalDeclarationStatement](#) method to skip rewriting these forms of declarations:

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

The method indicates that no rewriting takes place by returning the `node` parameter unmodified. If neither of those `if` expressions are true, the node represents a possible declaration with initialization. Add these statements to extract the type name specified in the declaration and bind it using the [SemanticModel](#) field to obtain a type symbol:

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Now, add this statement to bind the initializer expression:

```
var initializerInfo = SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Finally, add the following `if` statement to replace the existing type name with the `var` keyword if the type of the initializer expression matches the type specified:

```
if (SymbolEqualityComparer.Default.Equals(variableType, initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

The conditional is required because the declaration may cast the initializer expression to a base class or interface. If that's desired, the types on the left and right-hand side of the assignment don't match. Removing the explicit type in these cases would change the semantics of a program. `var` is specified as an identifier rather than a keyword because `var` is a contextual keyword. The leading and trailing trivia (white space) are transferred from the old type name to the `var` keyword to maintain vertical white space and indentation. It's simpler to use `ReplaceNode` rather than `With*` to transform the [LocalDeclarationStatementSyntax](#) because the type name is actually the grandchild of the declaration statement.

You've finished the `TypeInferenceRewriter`. Now return to your `Program.cs` file to finish the example. Create a test [Compilation](#) and obtain the [SemanticModel](#) from it. Use that [SemanticModel](#) to try your `TypeInferenceRewriter`. You'll do this step last. In the meantime declare a placeholder variable representing your test compilation:

```
Compilation test = CreateTestCompilation();
```

After pausing a moment, you should see an error squiggle appear reporting that no `CreateTestCompilation` method exists. Press **Ctrl+Period** to open the light-bulb and then press Enter to invoke the **Generate Method Stub** command. This command will generate a method stub for the `CreateTestCompilation` method in the `Program` class. You'll come back to fill in this method later:

The screenshot shows a Visual Studio code editor with the following code:

```
6  namespace TransformationCS
7  {
8  class Program
9  {
10     static void Main(string[] args)
11     {
12         Compilation test = CreateTestCompilation();
13     }
14 }
15 }
```

A tooltip is displayed at the bottom right, showing a warning message:

CS0103 The name 'CreateTestCompilation' does not exist in the current context

Below the tooltip, a code completion suggestion is shown:

```
...
}
private static Compilation CreateTestCompilation()
{
    throw new NotImplementedException();
}
...
```

At the bottom of the tooltip, there is a "Preview changes" button.

Write the following code to iterate over each [SyntaxTree](#) in the test [Compilation](#). For each one, initialize a new [TypeInferenceRewriter](#) with the [SemanticModel](#) for that tree:

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Inside the `foreach` statement you created, add the following code to perform the transformation on each source tree. This code conditionally writes out the new transformed tree if any edits were made. Your rewriter should only modify a tree if it encounters one or more local variable declarations that could be simplified using type inference:

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

You should see squiggles under the `File.WriteAllText` code. Select the light bulb, and add the necessary `using System.IO;` statement.

You're almost done! There's one step left: creating a test [Compilation](#). Since you haven't been using type inference at all during this quickstart, it would have made a perfect test case. Unfortunately, creating a [Compilation](#) from a C# project file is beyond the scope of this walkthrough. But fortunately, if you've been following instructions carefully, there's hope. Replace the contents of the `CreateTestCompilation` method with the following code. It creates a test compilation that coincidentally matches the project described in this quickstart:

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
    CSharpSyntaxTree.ParseText(programText)
        .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
    CSharpSyntaxTree.ParseText(rewriterText)
        .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
    MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
    MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Cross your fingers and run the project. In Visual Studio, choose **Debug > Start Debugging**. You should be prompted by Visual Studio that the files in your project have changed. Click "**Yes to All**" to reload the modified files. Examine them to observe your awesomeness. Note how much cleaner the code looks without all those explicit and redundant type specifiers.

Congratulations! You've used the **Compiler APIs** to write your own refactoring that searches all files in a C# project for certain syntactic patterns, analyzes the semantics of source code that matches those patterns, and transforms it. You're now officially a refactoring author!

Tutorial: Write your first analyzer and code fix

3/27/2021 • 23 minutes to read • [Edit Online](#)

The .NET Compiler Platform SDK provides the tools you need to create custom diagnostics (analyzers), code fixes, code refactoring, and diagnostic suppressors that target C# or Visual Basic code. An **analyzer** contains code that recognizes violations of your rule. Your **code fix** contains the code that fixes the violation. The rules you implement can be anything from code structure to coding style to naming conventions and more. The .NET Compiler Platform provides the framework for running analysis as developers are writing code, and all the Visual Studio UI features for fixing code: showing squiggles in the editor, populating the Visual Studio Error List, creating the "light bulb" suggestions and showing the rich preview of the suggested fixes.

In this tutorial, you'll explore the creation of an **analyzer** and an accompanying **code fix** using the Roslyn APIs. An analyzer is a way to perform source code analysis and report a problem to the user. Optionally, a code fix can be associated with the analyzer to represent a modification to the user's source code. This tutorial creates an analyzer that finds local variable declarations that could be declared using the `const` modifier but are not. The accompanying code fix modifies those declarations to add the `const` modifier.

Prerequisites

- [Visual Studio 2019](#) version 16.8 or later

You'll need to install the .NET Compiler Platform SDK via the Visual Studio Installer:

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the Visual Studio Installer:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

There are several steps to creating and validating your analyzer:

1. Create the solution.
2. Register the analyzer name and description.
3. Report analyzer warnings and recommendations.
4. Implement the code fix to accept recommendations.
5. Improve the analysis through unit tests.

Create the solution

- In Visual Studio, choose **File > New > Project...** to display the New Project dialog.
- Under **Visual C# > Extensibility**, choose **Analyzer with code fix (.NET Standard)**.
- Name your project "**MakeConst**" and click **OK**.

Explore the analyzer template

The analyzer with code fix template creates five projects:

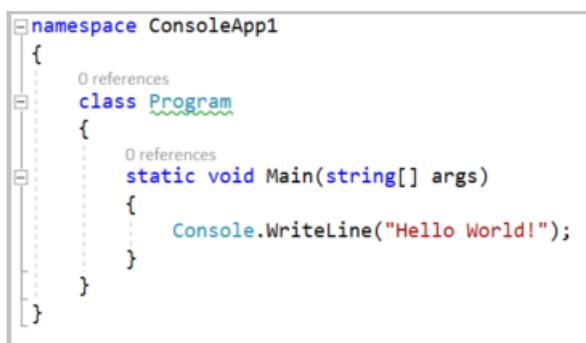
- **MakeConst**, which contains the analyzer.
- **MakeConst.CodeFixes**, which contains the code fix.
- **MakeConst.Package**, which is used to produce NuGet package for the analyzer and code fix.
- **MakeConst.Test**, which is a unit test project.
- **MakeConst.Vsix**, which is the default startup project that starts a second instance of Visual Studio that has loaded your new analyzer. Press F5 to start the VSIX project.

TIP

When you run your analyzer, you start a second copy of Visual Studio. This second copy uses a different registry hive to store settings. That enables you to differentiate the visual settings in the two copies of Visual Studio. You can pick a different theme for the experimental run of Visual Studio. In addition, don't roam your settings or login to your Visual Studio account using the experimental run of Visual Studio. That keeps the settings different.

In the second Visual Studio instance that you just started, create a new C# Console Application project (any target framework will work -- analyzers work at the source level.) Hover over the token with a wavy underline, and the warning text provided by an analyzer appears.

The template creates an analyzer that reports a warning on each type declaration where the type name contains lowercase letters, as shown in the following figure:



The template also provides a code fix that changes any type name containing lower case characters to all upper case. You can click on the light bulb displayed with the warning to see the suggested changes. Accepting the suggested changes updates the type name and all references to that type in the solution. Now that you've seen

the initial analyzer in action, close the second Visual Studio instance and return to your analyzer project.

You don't have to start a second copy of Visual Studio and create new code to test every change in your analyzer. The template also creates a unit test project for you. That project contains two tests. `TestMethod1` shows the typical format of a test that analyzes code without triggering a diagnostic. `TestMethod2` shows the format of a test that triggers a diagnostic, and then applies a suggested code fix. As you build your analyzer and code fix, you'll write tests for different code structures to verify your work. Unit tests for analyzers are much quicker than testing them interactively with Visual Studio.

TIP

Analyzer unit tests are a great tool when you know what code constructs should and shouldn't trigger your analyzer. Loading your analyzer in another copy of Visual Studio is a great tool to explore and find constructs you may not have thought about yet.

In this tutorial, you write an analyzer that reports to the user any local variable declarations that can be converted to local constants. For example, consider the following code:

```
int x = 0;  
Console.WriteLine(x);
```

In the code above, `x` is assigned a constant value and is never modified. It can be declared using the `const` modifier:

```
const int x = 0;  
Console.WriteLine(x);
```

The analysis to determine whether a variable can be made constant is involved, requiring syntactic analysis, constant analysis of the initializer expression and dataflow analysis to ensure that the variable is never written to. The .NET Compiler Platform provides APIs that make it easier to perform this analysis.

Create analyzer registrations

The template creates the initial `DiagnosticAnalyzer` class, in the `MakeConstAnalyzer.cs` file. This initial analyzer shows two important properties of every analyzer.

- Every diagnostic analyzer must provide a `[DiagnosticAnalyzer]` attribute that describes the language it operates on.
- Every diagnostic analyzer must derive (directly or indirectly) from the `DiagnosticAnalyzer` class.

The template also shows the basic features that are part of any analyzer:

1. Register actions. The actions represent code changes that should trigger your analyzer to examine code for violations. When Visual Studio detects code edits that match a registered action, it calls your analyzer's registered method.
2. Create diagnostics. When your analyzer detects a violation, it creates a diagnostic object that Visual Studio uses to notify the user of the violation.

You register actions in your override of `DiagnosticAnalyzer.Initialize(AnalysisContext)` method. In this tutorial, you'll visit **syntax nodes** looking for local declarations, and see which of those have constant values. If a declaration could be constant, your analyzer will create and report a diagnostic.

The first step is to update the registration constants and `Initialize` method so these constants indicate your "Make Const" analyzer. Most of the string constants are defined in the string resource file. You should follow that

practice for easier localization. Open the *Resources.resx* file for the *MakeConst* analyzer project. This displays the resource editor. Update the string resources as follows:

- Change `AnalyzerDescription` to "Variables that are not modified should be made constants."
- Change `AnalyzerMessageFormat` to "Variable '{0}' can be made constant".
- Change `AnalyzerTitle` to "Variable can be made constant".

When you have finished, the resource editor should appear as follow figure shows:

Name	Value	Comment
<code>AnalyzerDescription</code>	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
<code>AnalyzerMessageFormat</code>	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
► <code>AnalyzerTitle</code>	Variable can be made constant	The title of the diagnostic.
*		

The remaining changes are in the analyzer file. Open *MakeConstAnalyzer.cs* in Visual Studio. Change the registered action from one that acts on symbols to one that acts on syntax. In the `MakeConstAnalyzerAnalyzer.Initialize` method, find the line that registers the action on symbols:

```
context.RegisterSymbolAction>AnalyzeSymbol, SymbolKind.NamedType);
```

Replace it with the following line:

```
context.RegisterSyntaxNodeAction>AnalyzeNode, SyntaxKind.LocalDeclarationStatement);
```

After that change, you can delete the `AnalyzeSymbol` method. This analyzer examines `SyntaxKind.LocalDeclarationStatement`, not `SymbolKind.NamedType` statements. Notice that `AnalyzeNode` has red squiggles under it. The code you just added references an `AnalyzeNode` method that hasn't been declared. Declare that method using the following code:

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{}
```

Change the `Category` to "Usage" in *MakeConstAnalyzer.cs* as shown in the following code:

```
private const string Category = "Usage";
```

Find local declarations that could be const

It's time to write the first version of the `AnalyzeNode` method. It should look for a single local declaration that could be `const` but is not, like the following code:

```
int x = 0;
Console.WriteLine(x);
```

The first step is to find local declarations. Add the following code to `AnalyzeNode` in *MakeConstAnalyzer.cs*:

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

This cast always succeeds because your analyzer registered for changes to local declarations, and only local

declarations. No other node type triggers a call to your `AnalyzeNode` method. Next, check the declaration for any `const` modifiers. If you find them, return immediately. The following code looks for any `const` modifiers on the local declaration:

```
// make sure the declaration isn't already const:  
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))  
{  
    return;  
}
```

Finally, you need to check that the variable could be `const`. That means making sure it is never assigned after it is initialized.

You'll perform some semantic analysis using the `SyntaxNodeAnalysisContext`. You use the `context` argument to determine whether the local variable declaration can be made `const`. A `Microsoft.CodeAnalysis.SemanticModel` represents all of semantic information in a single source file. You can learn more in the article that covers [semantic models](#). You'll use the `Microsoft.CodeAnalysis.SemanticModel` to perform data flow analysis on the local declaration statement. Then, you use the results of this data flow analysis to ensure that the local variable isn't written with a new value anywhere else. Call the `GetDeclaredSymbol` extension method to retrieve the `ILocalSymbol` for the variable and check that it isn't contained with the `DataFlowAnalysis.WrittenOutside` collection of the data flow analysis. Add the following code to the end of the `AnalyzeNode` method:

```
// Perform data flow analysis on the local declaration.  
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);  
  
// Retrieve the local symbol for each variable in the local declaration  
// and ensure that it is not written outside of the data flow analysis region.  
VariableDeclaratorSyntax variable = localDeclaration.Declaration.Variables.Single();  
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);  
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))  
{  
    return;  
}
```

The code just added ensures that the variable isn't modified, and can therefore be made `const`. It's time to raise the diagnostic. Add the following code as the last line in `AnalyzeNode`:

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),  
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

You can check your progress by pressing F5 to run your analyzer. You can load the console application you created earlier and then add the following test code:

```
int x = 0;  
Console.WriteLine(x);
```

The light bulb should appear, and your analyzer should report a diagnostic. However, the light bulb still uses the template generated code fix, and tells you it can be made upper case. The next section explains how to write the code fix.

Write the code fix

An analyzer can provide one or more code fixes. A code fix defines an edit that addresses the reported issue. For the analyzer that you created, you can provide a code fix that inserts the `const` keyword:

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

The user chooses it from the light bulb UI in the editor and Visual Studio changes the code.

Open `CodeFixResources.resx` file and change `CodeFixTitle` to "Make constant".

Open the `MakeConstCodeFixProvider.cs` file added by the template. This code fix is already wired up to the Diagnostic ID produced by your diagnostic analyzer, but it doesn't yet implement the right code transform.

Next, delete the `MakeUppercaseAsync` method. It no longer applies.

All code fix providers derive from `CodeFixProvider`. They all override `CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext)` to report available code fixes. In `RegisterCodeFixesAsync`, change the ancestor node type you're searching for to a `LocalDeclarationStatementSyntax` to match the diagnostic:

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>()
.First();
```

Next, change the last line to register a code fix. Your fix will create a new document that results from adding the `const` modifier to an existing declaration:

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

You'll notice red squiggles in the code you just added on the symbol `MakeConstAsync`. Add a declaration for `MakeConstAsync` like the following code:

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{}
```

Your new `MakeConstAsync` method will transform the `Document` representing the user's source file into a new `Document` that now contains a `const` declaration.

You create a new `const` keyword token to insert at the front of the declaration statement. Be careful to first remove any leading trivia from the first token of the declaration statement and attach it to the `const` token. Add the following code to the `MakeConstAsync` method:

```

// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
    SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));

```

Next, add the `const` token to the declaration using the following code:

```

// Insert the const token into the modifiers list, creating a new modifiers list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);

```

Next, format the new declaration to match C# formatting rules. Formatting your changes to match existing code creates a better experience. Add the following statement immediately after the existing code:

```

// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal = newLocal.WithAdditionalAnnotations(Formatter.Annotation);

```

A new namespace is required for this code. Add the following `using` directive to the top of the file:

```
using Microsoft.CodeAnalysis.Formatting;
```

The final step is to make your edit. There are three steps to this process:

1. Get a handle to the existing document.
2. Create a new document by replacing the existing declaration with the new declaration.
3. Return the new document.

Add the following code to the end of the `MakeConstAsync` method:

```

// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);

```

Your code fix is ready to try. Press F5 to run the analyzer project in a second instance of Visual Studio. In the second Visual Studio instance, create a new C# Console Application project and add a few local variable declarations initialized with constant values to the Main method. You'll see that they are reported as warnings as below.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

You've made a lot of progress. There are squiggles under the declarations that can be made `const`. But there is

still work to do. This works fine if you add `const` to the declarations starting with `i`, then `j` and finally `k`. But, if you add the `const` modifier in a different order, starting with `k`, your analyzer creates errors: `k` can't be declared `const`, unless `i` and `j` are both already `const`. You've got to do more analysis to ensure you handle the different ways variables can be declared and initialized.

Build unit tests

Your analyzer and code fix work on a simple case of a single declaration that can be made `const`. There are numerous possible declaration statements where this implementation makes mistakes. You'll address these cases by working with the unit test library written by the template. It's much faster than repeatedly opening a second copy of Visual Studio.

Open the `MakeConstUnitTests.cs` file in the unit test project. The template created two tests that follow the two common patterns for an analyzer and code fix unit test. `TestMethod1` shows the pattern for a test that ensures the analyzer doesn't report a diagnostic when it shouldn't. `TestMethod2` shows the pattern for reporting a diagnostic and running the code fix.

The template uses [Microsoft.CodeAnalysis.Testing](#) packages for unit testing.

TIP

The testing library supports a special markup syntax, including the following:

- `[|text|]` : indicates that a diagnostic is reported for `text`. By default, this form may only be used for testing analyzers with exactly one `DiagnosticDescriptor` provided by `DiagnosticAnalyzer.SupportedDiagnostics`.
- `{|ExpectedDiagnosticId:text|}` : indicates that a diagnostic with Id `ExpectedDiagnosticId` is reported for `text`.

Replace the template tests in the `MakeConstUnitTest` class with the following test method:

```
[TestMethod]
public async Task LocalIntCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@""
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
},
@", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
");
    }
}
```

Run this test to make sure it passes. In Visual Studio, open the **Test Explorer** by selecting **Test > Windows > Test Explorer**. Then select **Run All**.

Create tests for valid declarations

As a general rule, analyzers should exit as quickly as possible, doing minimal work. Visual Studio calls registered analyzers as the user edits code. Responsiveness is a key requirement. There are several test cases for code that should not raise your diagnostic. Your analyzer already handles one of those tests, the case where a variable is assigned after being initialized. Add the following test method to represent that case:

```
[TestMethod]
public async Task VariableIsAssigned_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
}
```

This test passes as well. Next, add test methods for conditions you haven't handled yet:

- Declarations that are already `const`, because they are already const:

```
[TestMethod]
public async Task VariableIsAlreadyConst_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
}
```

- Declarations that have no initializer, because there is no value to use:

```

[TestMethod]
public async Task NoInitializer_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
}

```

- Declarations where the initializer is not a constant, because they can't be compile-time constants:

```

[TestMethod]
public async Task InitializerIsNotConstant_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
}

```

It can be even more complicated because C# allows multiple declarations as one statement. Consider the following test case string constant:

```

[TestMethod]
public async Task MultipleInitializers_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
}

```

The variable `i` can be made constant, but the variable `j` cannot. Therefore, this statement cannot be made a `const` declaration.

Run your tests again, and you'll see these new test cases fail.

Update your analyzer to ignore correct declarations

You need some enhancements to your analyzer's `AnalyzeNode` method to filter out code that matches these conditions. They are all related conditions, so similar changes will fix all these conditions. Make the following changes to `AnalyzeNode`:

- Your semantic analysis examined a single variable declaration. This code needs to be in a `foreach` loop that examines all the variables declared in the same statement.
- Each declared variable needs to have an initializer.
- Each declared variable's initializer must be a compile-time constant.

In your `AnalyzeNode` method, replace the original semantic analysis:

```
// Perform data flow analysis on the local declaration.  
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);  
  
// Retrieve the local symbol for each variable in the local declaration  
// and ensure that it is not written outside of the data flow analysis region.  
VariableDeclaratorSyntax variable = localDeclaration.Declaration.Variables.Single();  
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);  
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))  
{  
    return;  
}
```

with the following code snippet:

```
// Ensure that all variables in the local declaration have initializers that  
// are assigned with constant values.  
foreach (VariableDeclaratorSyntax variable in localDeclaration.Declaration.Variables)  
{  
    EqualsValueClauseSyntax initializer = variable.Initializer;  
    if (initializer == null)  
    {  
        return;  
    }  
  
    Optional<object> constantValue = context.SemanticModel.GetConstantValue(initializer.Value,  
context.CancellationToken);  
    if (!constantValue.HasValue)  
    {  
        return;  
    }  
}  
  
// Perform data flow analysis on the local declaration.  
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);  
  
foreach (VariableDeclaratorSyntax variable in localDeclaration.Declaration.Variables)  
{  
    // Retrieve the local symbol for each variable in the local declaration  
    // and ensure that it is not written outside of the data flow analysis region.  
    ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);  
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))  
    {  
        return;  
    }  
}
```

The first `foreach` loop examines each variable declaration using syntactic analysis. The first check guarantees that the variable has an initializer. The second check guarantees that the initializer is a constant. The second loop

has the original semantic analysis. The semantic checks are in a separate loop because it has a greater impact on performance. Run your tests again, and you should see them all pass.

Add the final polish

You're almost done. There are a few more conditions for your analyzer to handle. Visual Studio calls analyzers while the user is writing code. It's often the case that your analyzer will be called for code that doesn't compile. The diagnostic analyzer's `AnalyzeNode` method does not check to see if the constant value is convertible to the variable type. So, the current implementation will happily convert an incorrect declaration such as `int i = "abc"` to a local constant. Add a test method for this case:

```
[TestMethod]
public async Task DeclarationIsInvalid_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029: ""abc""|};
    }
}
");

}
```

In addition, reference types are not handled properly. The only constant value allowed for a reference type is `null`, except in this case of `System.String`, which allows string literals. In other words, `const string s = "abc"` is legal, but `const object s = "abc"` is not. This code snippet verifies that condition:

```
[TestMethod]
public async Task DeclarationIsNotString_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");

}
```

To be thorough, you need to add another test to make sure that you can create a constant declaration for a string. The following snippet defines both the code that raises the diagnostic, and the code after the fix has been applied:

```
[TestMethod]
public async Task StringCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@""
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @""
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
}
```

Finally, if a variable is declared with the `var` keyword, the code fix does the wrong thing and generates a `const var` declaration, which is not supported by the C# language. To fix this bug, the code fix must replace the `var` keyword with the inferred type's name:

```

[TestMethod]
public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", "@"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
}

[TestMethod]
public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", "@"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";;
    }
}
");
}

```

Fortunately, all of the above bugs can be addressed using the same techniques that you just learned.

To fix the first bug, first open `MakeConstAnalyzer.cs` and locate the foreach loop where each of the local declaration's initializers are checked to ensure that they're assigned with constant values. Immediately *before* the first foreach loop, call `context.SemanticModel.GetTypeInfo()` to retrieve detailed information about the declared type of the local declaration:

```

TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType = context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;

```

Then, inside your `foreach` loop, check each initializer to make sure it's convertible to the variable type. Add the following check after ensuring that the initializer is a constant:

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion = context.SemanticModel.ClassifyConversion(initializer.Value, valueType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

The next change builds upon the last one. Before the closing curly brace of the first foreach loop, add the following code to check the type of the local declaration when the constant is a string or null.

```
// Special cases:
// * If the constant value is a string, the type of the local declaration
//   must be System.String.
// * If the constant value is null, the type of the local declaration must
//   be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

You must write a bit more code in your code fix provider to replace the `var` keyword with the correct type name. Return to `MakeConstCodeFixProvider.cs`. The code you'll add does the following steps:

- Check if the declaration is a `var` declaration, and if it is:
- Create a new type for the inferred type.
- Make sure the type declaration is not an alias. If so, it is legal to declare `const var`.
- Make sure that `var` isn't a type name in this program. (If so, `const var` is legal).
- Simplify the full type name

That sounds like a lot of code. It's not. Replace the line that declares and initializes `newLocal` with the following code. It goes immediately after the initialization of `newModifiers`:

```

// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration = localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName, cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName, cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName = SyntaxFactory.ParseTypeName(type.ToString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName = typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclarationWithType(simplifiedTypeName);
        }
    }
}

// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal.WithModifiers(newModifiers)
    .WithDeclaration(variableDeclaration);

```

You'll need to add one `using` directive to use the [Simplifier](#) type:

```
using Microsoft.CodeAnalysis.Simplification;
```

Run your tests, and they should all pass. Congratulate yourself by running your finished analyzer. Press **Ctrl+F5** to run the analyzer project in a second instance of Visual Studio with the Roslyn Preview extension loaded.

- In the second Visual Studio instance, create a new C# Console Application project and add `int x = "abc";` to the Main method. Thanks to the first bug fix, no warning should be reported for this local variable declaration (though there's a compiler error as expected).
- Next, add `object s = "abc";` to the Main method. Because of the second bug fix, no warning should be reported.
- Finally, add another local variable that uses the `var` keyword. You'll see that a warning is reported and a suggestion appears beneath to the left.
- Move the editor caret over the squiggly underline and press **Ctrl+..** to display the suggested code fix. Upon selecting your code fix, note that the `var` keyword is now handled correctly.

Finally, add the following code:

```
int i = 2;  
int j = 32;  
int k = i + j;
```

After these changes, you get red squiggles only on the first two variables. Add `const` to both `i` and `j`, and you get a new warning on `k` because it can now be `const`.

Congratulations! You've created your first .NET Compiler Platform extension that performs on-the-fly code analysis to detect an issue and provides a quick fix to correct it. Along the way, you've learned many of the code APIs that are part of the .NET Compiler Platform SDK (Roslyn APIs). You can check your work against the [completed sample](#) in our samples GitHub repository.

Other resources

- [Get started with syntax analysis](#)
- [Get started with semantic analysis](#)

C# programming guide

4/6/2021 • 2 minutes to read • [Edit Online](#)

This section provides detailed information on key C# language features and features accessible to C# through .NET.

Most of this section assumes that you already know something about C# and general programming concepts. If you are a complete beginner with programming or with C#, you might want to visit the [Introduction to C# Tutorials](#) or [.NET In-Browser Tutorial](#), where no prior programming knowledge is required.

For information about specific keywords, operators, and preprocessor directives, see [C# Reference](#). For information about the C# Language Specification, see [C# Language Specification](#).

Program sections

[Inside a C# Program](#)

[Main\(\) and Command-Line Arguments](#)

Language Sections

[Statements, Expressions, and Operators](#)

[Types](#)

[Classes and Structs](#)

[Interfaces](#)

[Delegates](#)

[Arrays](#)

[Strings](#)

[Properties](#)

[Indexers](#)

[Events](#)

[Generics](#)

[Iterators](#)

[LINQ Query Expressions](#)

[Namespaces](#)

[Unsafe Code and Pointers](#)

[XML Documentation Comments](#)

Platform Sections

[Application Domains](#)

[Assemblies in .NET](#)

[Attributes](#)

[Collections](#)

[Exceptions and Exception Handling](#)

[File System and the Registry \(C# Programming Guide\)](#)

[Interoperability](#)

[Reflection](#)

See also

- [C# Reference](#)

Inside a C# program

3/6/2021 • 2 minutes to read • [Edit Online](#)

The section discusses the general structure of a C# program, and includes the standard "Hello, World!" example.

In this section

- [General Structure of a C# Program](#)
- [Identifier names](#)
- [C# Coding Conventions](#)

Related sections

- [Get Started with C#](#)
- [C# Programming Guide](#)
- [C# Reference](#)
- [Samples and tutorials](#)

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)

General Structure of a C# Program (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

C# programs can consist of one or more files. Each file can contain zero or more namespaces. A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces. The following is the skeleton of a C# program that contains all of these elements.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

Related Sections

For more information:

- [Classes](#)
- [Structs](#)
- [Namespaces](#)
- [Interfaces](#)

- [Delegates](#)

C# Language Specification

For more information, see [Basic concepts](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Inside a C# Program](#)
- [C# Reference](#)

Identifier names

2/26/2020 • 2 minutes to read • [Edit Online](#)

An **identifier** is the name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace. Valid identifiers must follow these rules:

- Identifiers must start with a letter, or `_`.
- Identifiers may contain Unicode letter characters, decimal digit characters, Unicode connecting characters, Unicode combining characters, or Unicode formatting characters. For more information on Unicode categories, see the [Unicode Category Database](#). You can declare identifiers that match C# keywords by using the `@` prefix on the identifier. The `@` is not part of the identifier name. For example, `@if` declares an identifier named `if`. These [verbatim identifiers](#) are primarily for interoperability with identifiers declared in other languages.

For a complete definition of valid identifiers, see the [Identifiers topic in the C# Language Specification](#).

Naming conventions

In addition to the rules, there are a number of identifier [naming conventions](#) used throughout the .NET APIs. By convention, C# programs use `PascalCase` for type names, namespaces, and all public members. In addition, the following conventions are common:

- Interface names start with a capital `I`.
- Attribute types end with the word `Attribute`.
- Enum types use a singular noun for non-flags, and a plural noun for flags.
- Identifiers should not contain two consecutive `_` characters. Those names are reserved for compiler generated identifiers.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Inside a C# Program](#)
- [C# Reference](#)
- [Classes](#)
- [Structure types](#)
- [Namespaces](#)
- [Interfaces](#)
- [Delegates](#)

C# Coding Conventions (C# Programming Guide)

4/2/2021 • 9 minutes to read • [Edit Online](#)

Coding conventions serve the following purposes:

- They create a consistent look to the code, so that readers can focus on content, not layout.
- They enable readers to understand the code more quickly by making assumptions based on previous experience.
- They facilitate copying, changing, and maintaining the code.
- They demonstrate C# best practices.

The guidelines in this article are used by Microsoft to develop samples and documentation.

Naming conventions

- In short examples that don't include [using directives](#), use namespace qualifications. If you know that a namespace is imported by default in a project, you don't have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.

```
var currentPerformanceCounterCategory = new System.Diagnostics.  
    PerformanceCounterCategory();
```

- You don't have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

Layout conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see [Options, Text Editor, C#, Formatting](#).
- Write only one statement per line.
- Write only one declaration per line.
- If continuation lines are not indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
if ((val1 > val2) && (val1 > val3))  
{  
    // Take appropriate action.  
}
```

Commenting conventions

- Place the comment on a separate line, not at the end of a line of code.
 - Begin comment text with an uppercase letter.
 - End comment text with a period.
 - Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

```
// The following declaration creates a query. It does not run  
// the query.
```

- Don't create formatted blocks of asterisks around comments.

Language guidelines

The following sections describe practices that the C# team follows to prepare code examples and samples.

String data type

- Use [string interpolation](#) to concatenate short strings, as shown in the following code.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- To append strings in loops, especially when you're working with large amounts of text, use a `StringBuilder` object.

Iimplicitly typed local variables

- Use [implicit typing](#) for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

```
var var1 = "This is clearly a string.";  
var var2 = 27;
```

- Don't use `var` when the type is not apparent from the right side of the assignment. Don't assume the type is clear from a method name. A variable type is considered clear if it's a `new` operator or an explicit cast.

```
int var3 = Convert.ToInt32(Console.ReadLine());  
int var4 = ExampleClass.ResultSoFar();
```

- Don't rely on the variable name to specify the type of the variable. It might not be correct. In the following example, the variable name `inputInt` is misleading. It's a string.

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Avoid the use of `var` in place of `dynamic`. Use `dynamic` when you want run-time type inference. For more information, see [Using type dynamic \(C# Programming Guide\)](#).
 - Use implicit typing to determine the type of the loop variable in `for` loops.

The following example uses implicit typing in a `for` statement.

- Don't use implicit typing to determine the type of the loop variable in `foreach` loops.

The following example uses explicit typing in a `foreach` statement.

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

NOTE

Be careful not to accidentally change a type of an element of the iterable collection. For example, it is easy to switch from [System.Linq.IQueryable](#) to [System.Collections.IEnumerable](#) in a `foreach` statement, which changes the execution of a query.

Unsigned data types

In general, use `int` rather than unsigned types. The use of `int` is common throughout C#, and it is easier to interact with other libraries when you use `int`.

Arrays

Use the concise syntax when you initialize arrays on the declaration line. In the following example, note that you can't use `var` instead of `string[]`.

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

If you use explicit instantiation, you can use `var`.

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

If you specify an array size, you have to initialize the elements one at a time.

```
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Delegates

Use `Func<>` and `Action<>` instead of defining delegate types. In a class, define the delegate method.

```
public static Action<string> ActionExample1 = x => Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x + y;
```

Call the method using the signature defined by the `Func<>` or `Action<>` delegate.

```
ActionExample1("string for x");

ActionExample2("string for x", "string for y");

Console.WriteLine($"The value is {FuncExample1("1")}");

Console.WriteLine($"The sum is {FuncExample2(1, 2)}");
```

If you create instances of a delegate type, use the concise syntax. In a class, define the delegate type and a method that has a matching signature.

```
public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

Create an instance of the delegate type and call it. The following declaration shows the condensed syntax.

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

The following declaration uses the full syntax.

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

try - catch and using statements in exception handling

- Use a `try-catch` statement for most exception handling.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

- Simplify your code by using the C# [using statement](#). If you have a [try-finally](#) statement in which the only code in the `finally` block is a call to the [Dispose](#) method, use a `using` statement instead.

In the following example, the `try - finally` statement only calls `Dispose` in the `finally` block.

```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

You can do the same thing with a `using` statement.

```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset2 = font2.GdiCharSet;
}
```

In C# 8 and later versions, use the new [using syntax](#) that doesn't require braces:

```
using Font font3 = new Font("Arial", 10.0f);
byte charset3 = font3.GdiCharSet;
```

`&&` and `||` operators

To avoid exceptions and increase performance by skipping unnecessary comparisons, use `&&` instead of `&` and `||` instead of `|` when you perform comparisons, as shown in the following example.

```
Console.WriteLine("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

If the divisor is 0, the second clause in the `if` statement would cause a run-time error. But the `&&` operator short-circuits when the first expression is false. That is, it doesn't evaluate the second expression. The `&` operator would evaluate both, resulting in a run-time error when `divisor` is 0.

`new` operator

- Use one of the concise forms of object instantiation, as shown in the following declarations. The second example shows syntax that is available starting in C# 9.

```
var instance1 = new ExampleClass();
```

```
ExampleClass instance2 = new();
```

The preceding declarations are equivalent to the following declaration.

```
ExampleClass instance2 = new ExampleClass();
```

- Use object initializers to simplify object creation, as shown in the following example.

```
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

The following example sets the same properties as the preceding example but doesn't use initializers.

```
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Event handling

If you're defining an event handler that you don't need to remove later, use a lambda expression.

```

public Form2()
{
    this.Click += (s, e) =>
    {
        MessageBox.Show(
            ((MouseEventArgs)e).Location.ToString());
    };
}

```

The lambda expression shortens the following traditional definition.

```

public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}

```

Static members

Call `static` members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Don't qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

LINQ queries

- Use meaningful names for query variables. The following example uses `seattleCustomers` for customers who are located in Seattle.

```

var seattleCustomers = from customer in customers
                      where customer.City == "Seattle"
                      select customer.Name;

```

- Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

```

var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { Customer = customer, Distributor = distributor };

```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as `Name` and `ID` in the result, rename them to clarify that `Name` is the name of a customer, and `ID` is the ID of a distributor.

```

var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { CustomerName = customer.Name, DistributorID = distributor.ID };

```

- Use implicit typing in the declaration of query variables and range variables.

```
var seattleCustomers = from customer in customers
    where customer.City == "Seattle"
    select customer.Name;
```

- Align query clauses under the `from` clause, as shown in the previous examples.
- Use `where` clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

```
var seattleCustomers2 = from customer in customers
    where customer.City == "Seattle"
    orderby customer.Name
    select customer;
```

- Use multiple `from` clauses instead of a `join` clause to access inner collections. For example, a collection of `Student` objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

```
var scoreQuery = from student in students
    from score in student.Scores
    where score > 90
    select new { Last = student.LastName, score };
```

Security

Follow the guidelines in [Secure Coding Guidelines](#).

See also

- [.NET runtime coding guidelines](#)
- [Visual Basic Coding Conventions](#)
- [Secure Coding Guidelines](#)

Main() and command-line arguments (C# Programming Guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

The `Main` method is the entry point of a C# application. (Libraries and services do not require a `Main` method as an entry point.) When the application is started, the `Main` method is the first method that is invoked.

There can only be one entry point in a C# program. If you have more than one class that has a `Main` method, you must compile your program with the `StartupObject` compiler option to specify which `Main` method to use as the entry point. For more information, see [StartupObject \(C# Compiler Options\)](#).

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

Starting in C# 9, you can omit the `Main` method, and write C# statements as if they were in the `Main` method, as in the following example:

```
using System;

Console.WriteLine("Hello World!");
```

For information about how to write application code with an implicit entry point method, see [Top-level statements](#).

Overview

- The `Main` method is the entry point of an executable program; it is where the program control starts and ends.
- `Main` is declared inside a class or struct. `Main` must be `static` and it need not be `public`. (In the earlier example, it receives the default access of `private`.) The enclosing class or struct is not required to be `static`.
- `Main` can either have a `void`, `int`, or, starting with C# 7.1, `Task`, or `Task<int>` return type.
- If and only if `Main` returns a `Task` or `Task<int>`, the declaration of `Main` may include the `async` modifier. Note that this specifically excludes an `async void Main` method.
- The `Main` method can be declared with or without a `string[]` parameter that contains command-line arguments. When using Visual Studio to create Windows applications, you can add the parameter manually or else use the `GetCommandLineArgs()` method to obtain the `command-line arguments`. Parameters are read as zero-indexed command-line arguments. Unlike C and C++, the name of the program is not treated as the first command-line argument in the `args` array, but it is the first element of the `GetCommandLineArgs()` method.

The following is a list of valid `Main` signatures:

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

The preceding examples all use the `public` accessor modifier. That is typical, but not required.

The addition of `async` and `Task`, `Task<int>` return types simplifies program code when console applications need to start and `await` asynchronous operations in `Main`.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Methods](#)
- [Inside a C# Program](#)

Command-Line Arguments (C# Programming Guide)

3/13/2021 • 3 minutes to read • [Edit Online](#)

You can send arguments to the `Main` method by defining the method in one of the following ways:

MAIN METHOD CODE	MAIN SIGNATURE
No return value, no use of <code>await</code>	<code>static void Main(string[] args)</code>
Return value, no use of <code>await</code>	<code>static int Main(string[] args)</code>
No return value, uses <code>await</code>	<code>static async Task Main(string[] args)</code>
Return value, uses <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

If the arguments are not used, you can omit `args` from the method signature for slightly simpler code:

MAIN METHOD CODE	MAIN SIGNATURE
No return value, no use of <code>await</code>	<code>static void Main()</code>
Return value, no use of <code>await</code>	<code>static int Main()</code>
No return value, uses <code>await</code>	<code>static async Task Main()</code>
Return value, uses <code>await</code>	<code>static async Task<int> Main()</code>

NOTE

To enable command-line arguments in the `Main` method in a Windows Forms application, you must manually modify the signature of `Main` in `program.cs`. The code generated by the Windows Forms designer creates a `Main` without an input parameter. You can also use `Environment.CommandLine` or `Environment.GetCommandLineArgs` to access the command-line arguments from any point in a console or Windows application.

The parameter of the `Main` method is a `String` array that represents the command-line arguments. Usually you determine whether arguments exist by testing the `Length` property, for example:

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

TIP

The `args` array cannot be null. So, it's safe to access the `Length` property without null checking.

You can also convert the string arguments to numeric types by using the [Convert](#) class or the [Parse](#) method. For example, the following statement converts the `string` to a `long` number by using the [Parse](#) method:

```
long num = Int64.Parse(args[0]);
```

It is also possible to use the C# type `long`, which aliases `Int64`:

```
long num = long.Parse(args[0]);
```

You can also use the `Convert` class method `ToInt64` to do the same thing:

```
long num = Convert.ToInt64(s);
```

For more information, see [Parse](#) and [Convert](#).

Example

The following example shows how to use command-line arguments in a console application. The application takes one argument at run time, converts the argument to an integer, and calculates the factorial of the number. If no arguments are supplied, the application issues a message that explains the correct usage of the program.

To compile and run the application from a command prompt, follow these steps:

1. Paste the following code into any text editor, and then save the file as a text file with the name *Factorial.cs*.

```

// Add a using directive for System if the directive isn't already present.

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively.
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied.
        if (args.Length == 0)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (!test)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.

```

2. From the **Start** screen or **Start** menu, open a Visual Studio **Developer Command Prompt** window, and then navigate to the folder that contains the file that you just created.
3. Enter the following command to compile the application.

```
csc Factorial.cs
```

If your application has no compilation errors, an executable file that's named *Factorial.exe* is created.

4. Enter the following command to calculate the factorial of 3:

```
Factorial 3
```

5. The command produces this output:

```
The factorial of 3 is 6.
```

NOTE

When running an application in Visual Studio, you can specify command-line arguments in the [Debug Page](#), [Project Designer](#).

See also

- [System.Environment](#)
- [C# Programming Guide](#)
- [Main\(\) and Command-Line Arguments](#)
- [How to display command line arguments](#)
- [Main\(\) Return Values](#)
- [Classes](#)

How to display command-line arguments (C# Programming Guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Arguments provided to an executable on the command line are accessible in [top-level statements](#) or through an optional parameter to `Main`. The arguments are provided in the form of an array of strings. Each element of the array contains one argument. White-space between arguments is removed. For example, consider these command-line invocations of a fictitious executable:

INPUT ON COMMAND LINE	ARRAY OF STRINGS PASSED TO MAIN
<code>executable.exe a b c</code>	"a" "b" "c"
<code>executable.exe one two</code>	"one" "two"
<code>executable.exe "one two" three</code>	"one two" "three"

NOTE

When you are running an application in Visual Studio, you can specify command-line arguments in the [Debug Page](#), [Project Designer](#).

Example

This example displays the command-line arguments passed to a command-line application. The output shown is for the first entry in the table above.

```
class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements.
        Console.WriteLine($"parameter count = {args.Length}");

        for (int i = 0; i < args.Length; i++)
        {
            Console.WriteLine($"Arg[{i}] = [{args[i]}]");
        }
    }
}

/* Output (assumes 3 cmd line args):
   parameter count = 3
   Arg[0] = [a]
   Arg[1] = [b]
   Arg[2] = [c]
*/
```

See also

- [C# Programming Guide](#)
- [Main\(\) and Command-Line Arguments](#)
- [Main\(\) Return Values](#)

Main() return values (C# Programming Guide)

3/13/2021 • 3 minutes to read • [Edit Online](#)

You can return an `int` from the `Main` method by defining the method in one of the following ways:

MAIN METHOD CODE	MAIN SIGNATURE
No use of <code>args</code> or <code>await</code>	<code>static int Main()</code>
Uses <code>args</code> , no use of <code>await</code>	<code>static int Main(string[] args)</code>
No use of <code>args</code> , uses <code>await</code>	<code>static async Task<int> Main()</code>
Uses <code>args</code> and <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

If the return value from `Main` is not used, returning `void` or `Task` allows for slightly simpler code.

MAIN METHOD CODE	MAIN SIGNATURE
No use of <code>args</code> or <code>await</code>	<code>static void Main()</code>
Uses <code>args</code> , no use of <code>await</code>	<code>static void Main(string[] args)</code>
No use of <code>args</code> , uses <code>await</code>	<code>static async Task Main()</code>
Uses <code>args</code> and <code>await</code>	<code>static async Task Main(string[] args)</code>

However, returning `int` or `Task<int>` enables the program to communicate status information to other programs or scripts that invoke the executable file.

The following example shows how the exit code for the process can be accessed.

Example

This example uses [.NET Core](#) command-line tools. If you are unfamiliar with .NET Core command-line tools, you can learn about them in this [get-started article](#).

Modify the `Main` method in `program.cs` as follows:

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

When a program is executed in Windows, any value returned from the `Main` function is stored in an environment variable. This environment variable can be retrieved using `ERRORLEVEL` from a batch file, or

```
$LastExitCode
```

 from PowerShell.

You can build the application using the [dotnet CLI](#) `dotnet build` command.

Next, create a PowerShell script to run the application and display the result. Paste the following code into a text file and save it as `test.ps1` in the folder that contains the project. Run the PowerShell script by typing `test.ps1` at the PowerShell prompt.

Because the code returns zero, the batch file will report success. However, if you change `MainReturnValTest.cs` to return a non-zero value and then recompile the program, subsequent execution of the PowerShell script will report failure.

```
dotnet run
```

```
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

Sample output

```
Execution succeeded
Return value = 0
```

Async Main return values

Async Main return values move the boilerplate code necessary for calling asynchronous methods in `Main` to code generated by the compiler. Previously, you would need to write this construct to call asynchronous code and ensure your program ran until the asynchronous operation completed:

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

Now, this can be replaced by:

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

The advantage of the new syntax is that the compiler always generates the correct code.

Compiler-generated code

When the application entry point returns a `Task` or `Task<int>`, the compiler generates a new entry point that calls the entry point method declared in the application code. Assuming that this entry point is called `$GeneratedMain`, the compiler generates the following code for these entry points:

- `static Task Main()` results in the compiler emitting the equivalent of

```
private static void $GeneratedMain() => Main().GetAwaiter().GetResult();
```
- `static Task Main(string[])` results in the compiler emitting the equivalent of

```
private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();
```
- `static Task<int> Main()` results in the compiler emitting the equivalent of

```
private static int $GeneratedMain() => Main().GetAwaiter().GetResult();
```
- `static Task<int> Main(string[])` results in the compiler emitting the equivalent of

```
private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();
```

NOTE

If the examples used `async` modifier on the `Main` method, the compiler would generate the same code.

See also

- [C# Programming Guide](#)
- [C# Reference](#)
- [Main\(\) and Command-Line Arguments](#)
- [How to display command line arguments](#)

Top-level statements (C# Programming Guide)

3/23/2021 • 3 minutes to read • [Edit Online](#)

Starting in C# 9, you don't have to explicitly include a `Main` method in a console application project. Instead, you can use the *top-level statements* feature to minimize the code you have to write. In this case, the compiler generates a class and `Main` method entry point for the application.

Here's a `Program.cs` file that is a complete C# program in C# 9:

```
using System;

Console.WriteLine("Hello World!");
```

Top-level statements let you write simple programs for small utilities such as Azure Functions and GitHub Actions. They also make it simpler for new C# programmers to get started learning and writing code.

The following sections explain the rules on what you can and can't do with top-level statements.

Only one top-level file

An application must have only one entry point, so a project can have only one file with top-level statements. Putting top-level statements in more than one file in a project results in the following compiler error:

CS8802 Only one compilation unit can have top-level statements.

A project can have any number of additional source code files that don't have top-level statements.

No other entry points

You can write a `Main` method explicitly, but it can't function as an entry point. The compiler issues the following warning:

CS7022 The entry point of the program is global code; ignoring 'Main()' entry point.

In a project with top-level statements, you can't use the `-main` compiler option to select the entry point, even if the project has one or more `Main` methods.

using directives

If you include using directives, they must come first in the file, as in this example:

```
using System;

Console.WriteLine("Hello World!");
```

Global namespace

Top-level statements are implicitly in the global namespace.

Namespaces and type definitions

A file with top-level statements can also contain namespaces and type definitions, but they must come after the top-level statements. For example:

```
using System;

MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

args

Top-level statements can reference the `args` variable to access any command-line arguments that were entered. The `args` variable is never null but its `Length` is zero if no command-line arguments were provided. For example:

```
using System;

if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

await

You can call an async method by using `await`. For example:

```
using System;
using System.Threading.Tasks;

Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

Exit code for the process

To return an `int` value when the application ends, use the `return` statement as you would in a `Main` method that returns an `int`. For example:

```
using System;

int returnValue = int.Parse(Console.ReadLine());
return returnValue;
```

Implicit entry point method

The compiler generates a method to serve as the program entry point for a project with top-level statements. The name of this method isn't actually `Main`, it's an implementation detail that your code can't reference directly. The signature of the method depends on whether the top-level statements contain the `await` keyword or the `return` statement. The following table shows what the method signature would look like, using the method name `Main` in the table for convenience.

TOP-LEVEL CODE CONTAINS	IMPLICIT <code>MAIN</code> SIGNATURE
<code>await</code> and <code>return</code>	<code>static async Task<int> Main(string[] args)</code>
<code>await</code>	<code>static async Task Main(string[] args)</code>
<code>return</code>	<code>static int Main(string[] args)</code>
No <code>await</code> or <code>return</code>	<code>static void Main(string[] args)</code>

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

[Top-level statements](#)

See also

- [C# Programming Guide](#)
- [Methods](#)
- [Inside a C# Program](#)

Programming Concepts (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section explains programming concepts in the C# language.

In This Section

TITLE	DESCRIPTION
Assemblies in .NET	Describes how to create and use assemblies.
Asynchronous Programming with <code>async</code> and <code>await</code> (C#)	Describes how to write asynchronous solutions by using the <code>async</code> and <code>await</code> keywords in C#. Includes a walkthrough.
Attributes (C#)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Collections (C#)	Describes some of the types of collections provided by .NET. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (C#)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (C#)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (C#)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (C#)	Discusses the powerful query capabilities in the language syntax of C#, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Reflection (C#)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (C#)	Describes key concepts in binary, XML, and SOAP serialization.

Related Sections

Performance Tips	Discusses several basic rules that may help you increase the performance of your application.
----------------------------------	---

Asynchronous programming with `async` and `await`

3/18/2021 • 15 minutes to read • [Edit Online](#)

The [Task asynchronous programming model \(TAP\)](#) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a [Task](#) that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the `async` and `await` keywords make it easier to reason about code, that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
        }
    }
}
```

```

Console.WriteLine("bacon is ready");

Toast toast = ToastBread(2);
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");

Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
}

private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");

private static Toast ToastBread(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}

private static Bacon FryBacon(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the pan");
    Console.WriteLine("cooking first side of bacon...");
    Task.Delay(3000).Wait();
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}

private static Egg FryEggs(int howMany)
{
    Console.WriteLine("Warming the egg pan...");
    Task.Delay(3000).Wait();
    Console.WriteLine($"cracking {howMany} eggs");
    Console.WriteLine("cooking the eggs ...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put eggs on plate");

    return new Egg();
}

private static Coffee PourCoffee()
{
    Console.WriteLine("Pouring coffee");
    return new Coffee();
}

```

```
    }  
}  
}
```



The synchronously prepared breakfast, took roughly 30 minutes because the total is the sum of each individual task.

NOTE

The `Coffee`, `Egg`, `Bacon`, `Toast`, and `Juice` classes are empty. They are simply marker classes for the purpose of demonstration, contain no properties, and serve no other purpose.

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that its step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

IMPORTANT

The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

TIP

The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the [final version](#) later in this article.

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The `System.Threading.Tasks.Task` and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd actually create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then await for something else that requires your attention.

You start a task and hold on to the `Task` object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");

Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

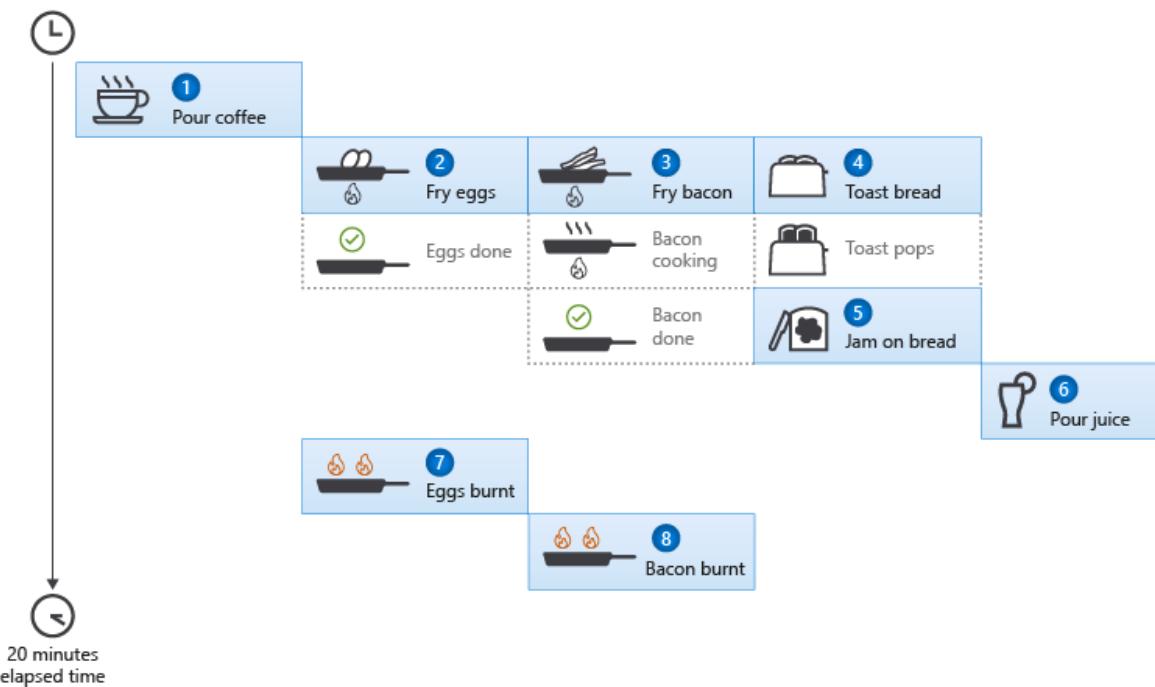
```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```



The asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests of different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

IMPORTANT

The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use `Task` or `Task<TResult>` objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts

the bread, then adds butter and jam. This method returns a `Task<TResult>` that represents the composition of those three operations. The main block of code now becomes:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is `awaited`. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

NOTE

You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

```
Pouring coffee
coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
flipping a slice of bacon
flipping a slice of bacon
flipping a slice of bacon
cooking the second side of bacon...
cracking 2 eggs
cooking the eggs ...
Put bacon on plate
Put eggs on plate
eggs are ready
bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

Notice that there's quite a few tasks completing between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is *faulted*. The Task object holds the exception thrown in the [Task.Exception](#) property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the [Task](#). The [Task.Exception](#) property is an [System.AggregateException](#) because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the [AggregateException.InnerExceptions](#) collection. If that [Exception](#) property is null, a new [AggregateException](#) is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the [Exception](#) property contains exactly one exception. When code [awaits](#) a faulted task, the first exception in the [AggregateException.InnerExceptions](#) collection is rethrown. That's why the output from this example shows an [InvalidOperationException](#) instead of an [AggregateException](#). Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the [Exception](#) property in your code when your scenario may generate multiple exceptions.

Before going on, comment out these two lines in your [ToastBreadAsync](#) method. You don't want to start another fire:

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is `WhenAll`, which returns a `Task` that completes when all the tasks in its argument list have completed, as shown in the following code:

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use `WhenAny`, which returns a `Task<Task>` that completes when any of its arguments completes. You can await the returned task, knowing that it has already finished. The following code shows how you could use `WhenAny` to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

After all those changes, the final version of the code looks like this:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
            }
        }
    }
}
```

```

        }

        else if (finishedTask == baconTask)
        {
            Console.WriteLine("bacon is ready");
        }
        else if (finishedTask == toastTask)
        {
            Console.WriteLine("toast is ready");
        }
        breakfastTasks.Remove(finishedTask);
    }

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}

private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");

private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(3000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}

private static async Task<Bacon> FryBaconAsync(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the pan");
    Console.WriteLine("cooking first side of bacon...");
    await Task.Delay(3000);
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    await Task.Delay(3000);
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}

private static async Task<Egg> FryEggsAsync(int howMany)
{

```

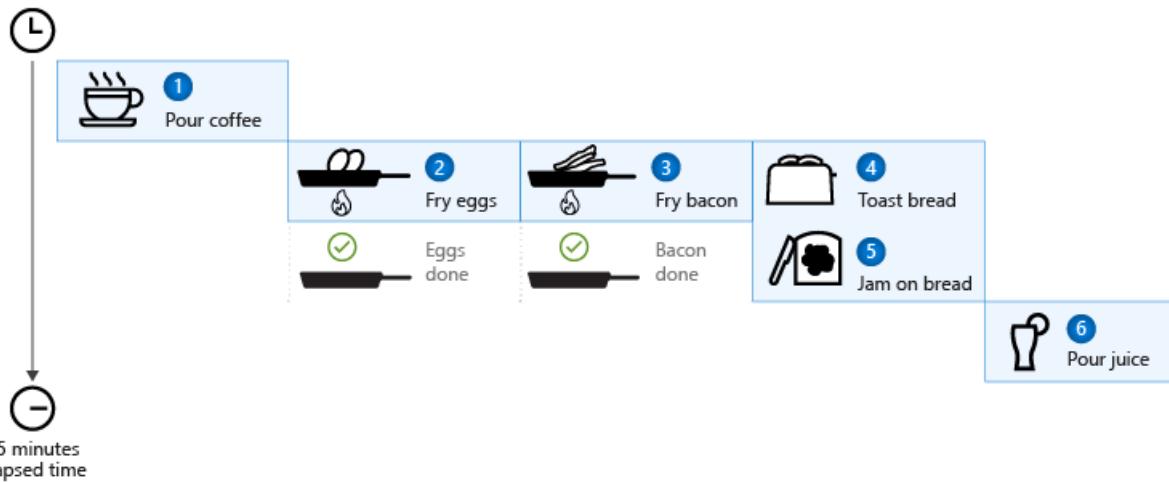
```

        Console.WriteLine("Warming the egg pan...");
        await Task.Delay(3000);
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
        await Task.Delay(3000);
        Console.WriteLine("Put eggs on plate");

        return new Egg();
    }

    private static Coffee PourCoffee()
    {
        Console.WriteLine("Pouring coffee");
        return new Coffee();
    }
}
}

```



The final version of the asynchronously prepared breakfast took roughly 15 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only take action when it was needed.

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Next steps

[Explore real world scenarios for asynchronous programs](#)

Asynchronous programming

3/27/2021 • 10 minutes to read • [Edit Online](#)

If you have any I/O-bound needs (such as requesting data from a network, accessing a database, or reading and writing to a file system), you'll want to utilize asynchronous programming. You could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.

C# has a language-level asynchronous programming model, which allows for easily writing asynchronous code without having to juggle callbacks or conform to a library that supports asynchrony. It follows what is known as the [Task-based Asynchronous Pattern \(TAP\)](#).

Overview of the asynchronous model

The core of async programming is the `Task` and `Task<T>` objects, which model asynchronous operations. They are supported by the `async` and `await` keywords. The model is fairly simple in most cases:

- For I/O-bound code, you await an operation that returns a `Task` or `Task<T>` inside of an `async` method.
- For CPU-bound code, you await an operation that is started on a background thread with the `Task.Run` method.

The `await` keyword is where the magic happens. It yields control to the caller of the method that performed `await`, and it ultimately allows a UI to be responsive or a service to be elastic. While [there are ways](#) to approach async code other than `async` and `await`, this article focuses on the language-level constructs.

I/O-bound example: Download data from a web service

You may need to download some data from a web service when a button is pressed but don't want to block the UI thread. It can be accomplished like this:

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

The code expresses the intent (downloading data asynchronously) without getting bogged down in interacting with `Task` objects.

CPU-bound example: Perform a calculation for a game

Say you're writing a mobile game where pressing a button can inflict damage on many enemies on the screen. Performing the damage calculation can be expensive, and doing it on the UI thread would make the game appear to pause as the calculation is performed!

The best way to handle this is to start a background thread, which does the work using `Task.Run`, and await its result using `await`. This allows the UI to feel smooth as the work is being done.

```

private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};

```

This code clearly expresses the intent of the button's click event, it doesn't require managing a background thread manually, and it does so in a non-blocking way.

What happens under the covers

There are many moving pieces where asynchronous operations are concerned. If you're curious about what's happening underneath the covers of `Task` and `Task<T>`, see the [Async in-depth](#) article for more information.

On the C# side of things, the compiler transforms your code into a state machine that keeps track of things like yielding execution when an `await` is reached and resuming execution when a background job has finished.

For the theoretically inclined, this is an implementation of the [Promise Model of asynchrony](#).

Key pieces to understand

- Async code can be used for both I/O-bound and CPU-bound code, but differently for each scenario.
- Async code uses `Task<T>` and `Task`, which are constructs used to model work being done in the background.
- The `async` keyword turns a method into an async method, which allows you to use the `await` keyword in its body.
- When the `await` keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.
- `await` can only be used inside an async method.

Recognize CPU-bound and I/O-bound work

The first two examples of this guide showed how you could use `async` and `await` for I/O-bound and CPU-bound work. It's key that you can identify when a job you need to do is I/O-bound or CPU-bound because it can greatly affect the performance of your code and could potentially lead to misusing certain constructs.

Here are two questions you should ask before you write any code:

1. Will your code be "waiting" for something, such as data from a database?

If your answer is "yes", then your work is **I/O-bound**.

2. Will your code be performing an expensive computation?

If you answered "yes", then your work is **CPU-bound**.

If the work you have is **I/O-bound**, use `async` and `await` *without* `Task.Run`. You *should not* use the Task Parallel Library. The reason for this is outlined in [Async in Depth](#).

If the work you have is **CPU-bound** and you care about responsiveness, use `async` and `await`, but spawn off the work on another thread *with* `Task.Run`. If the work is appropriate for concurrency and parallelism, also consider using the [Task Parallel Library](#).

Additionally, you should always measure the execution of your code. For example, you may find yourself in a situation where your CPU-bound work is not costly enough compared with the overhead of context switches when multithreading. Every choice has its tradeoff, and you should pick the correct tradeoff for your situation.

More examples

The following examples demonstrate various ways you can write async code in C#. They cover a few different scenarios you may come across.

Extract data from a network

This snippet downloads the HTML from the homepage at <https://dotnetfoundation.org> and counts the number of times the string ".NET" occurs in the HTML. It uses ASP.NET to define a Web API controller method, which performs this task and returns the number.

NOTE

If you plan on doing HTML parsing in production code, don't use regular expressions. Use a parsing library instead.

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.\.NET").Count;
}
```

Here's the same scenario written for a Universal Windows App, which performs the same task when a Button is pressed:

```

private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}

```

Wait for multiple tasks to complete

You may find yourself in a situation where you need to retrieve multiple pieces of data concurrently. The `Task` API contains two methods, `Task.WhenAll` and `Task.WhenAny`, that allow you to write asynchronous code that performs a non-blocking wait on multiple background jobs.

This example shows how you might grab `User` data for a set of `userId`s.

```

public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}

```

Here's another way to write this more succinctly, using LINQ:

```

public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}

```

Although it's less code, use caution when mixing LINQ with asynchronous code. Because LINQ uses deferred (lazy) execution, async calls won't happen immediately as they do in a `foreach` loop unless you force the generated sequence to iterate with a call to `.ToList()` or `.ToArray()`.

Important info and advice

With async programming, there are some details to keep in mind that can prevent unexpected behavior.

- `async` methods need to have an `await` keyword in their body or they will never yield!

This is important to keep in mind. If `await` is not used in the body of an `async` method, the C# compiler generates a warning, but the code compiles and runs as if it were a normal method. This is incredibly inefficient, as the state machine generated by the C# compiler for the `async` method is not accomplishing anything.

- Add "Async" as the suffix of every `async` method name you write.

This is the convention used in .NET to more easily differentiate synchronous and asynchronous methods. Certain methods that aren't explicitly called by your code (such as event handlers or web controller methods) don't necessarily apply. Because they are not explicitly called by your code, being explicit about their naming isn't as important.

- `async void` should only be used for event handlers.

`async void` is the only way to allow asynchronous event handlers to work because events do not have return types (thus cannot make use of `Task` and `Task<T>`). Any other use of `async void` does not follow the TAP model and can be challenging to use, such as:

- Exceptions thrown in an `async void` method can't be caught outside of that method.
- `async void` methods are difficult to test.
- `async void` methods can cause bad side effects if the caller isn't expecting them to be `async`.

- Tread carefully when using `async` lambdas in LINQ expressions

Lambda expressions in LINQ use deferred execution, meaning code could end up executing at a time when you're not expecting it to. The introduction of blocking tasks into this can easily result in a deadlock if not written correctly. Additionally, the nesting of asynchronous code like this can also make it more difficult to reason about the execution of the code. Async and LINQ are powerful but should be used together as carefully and clearly as possible.

- Write code that awaits Tasks in a non-blocking manner

Blocking the current thread as a means to wait for a `Task` to complete can result in deadlocks and blocked context threads and can require more complex error-handling. The following table provides guidance on how to deal with waiting for tasks in a non-blocking way:

USE THIS...	INSTEAD OF THIS...	WHEN WISHING TO DO THIS...
<code>await</code>	<code>Task.Wait</code> or <code>Task.Result</code>	Retrieving the result of a background task
<code>await Task.WhenAny</code>	<code>Task.WaitAny</code>	Waiting for any task to complete
<code>await Task.WhenAll</code>	<code>Task.WaitAll</code>	Waiting for all tasks to complete
<code>await Task.Delay</code>	<code>Thread.Sleep</code>	Waiting for a period of time

- Consider using `ValueTask` where possible

Returning a `Task` object from `async` methods can introduce performance bottlenecks in certain paths. `Task` is a reference type, so using it means allocating an object. In cases where a method declared with the `async` modifier returns a cached result or completes synchronously, the extra allocations can become a significant time cost in performance critical sections of code. It can become costly if those allocations occur in tight loops. For more information, see [generalized async return types](#).

- Consider using `ConfigureAwait(false)`

A common question is, "when should I use the `Task.ConfigureAwait(Boolean)` method?". The method allows for a `Task` instance to configure its awainer. This is an important consideration and setting it incorrectly could potentially have performance implications and even deadlocks. For more information on `ConfigureAwait`, see the [ConfigureAwait FAQ](#).

- Write less stateful code

Don't depend on the state of global objects or the execution of certain methods. Instead, depend only on the return values of methods. Why?

- Code will be easier to reason about.
- Code will be easier to test.
- Mixing `async` and synchronous code is far simpler.
- Race conditions can typically be avoided altogether.
- Depending on return values makes coordinating `async` code simple.
- (Bonus) it works really well with dependency injection.

A recommended goal is to achieve complete or near-complete [Referential Transparency](#) in your code. Doing so will result in a predictable, testable, and maintainable codebase.

Other resources

- [Async in-depth](#) provides more information about how Tasks work.
- [The Task asynchronous programming model \(C#\)](#).
- Lucian Wischik's [Six Essential Tips for Async](#) is a wonderful resource for `async` programming.

Task asynchronous programming model

11/2/2020 • 13 minutes to read • [Edit Online](#)

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

C# 5 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher, .NET Core, and the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

Async improves responsiveness

Asynchrony is essential for activities that are potentially blocking, such as web access. Access to a web resource sometimes is slow or delayed. If such an activity is blocked in a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from .NET and the Windows Runtime contain methods that support async programming.

APPLICATION AREA	.NET TYPES WITH ASYNC METHODS	WINDOWS RUNTIME TYPES WITH ASYNC METHODS
Web access	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Working with files	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
Working with images		MediaCapture BitmapEncoder BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations	

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional

asynchronous programming but with much less effort from the developer.

Async methods are easy to write

The `async` and `await` keywords in C# are the heart of async programming. By using those two keywords, you can use resources in .NET Framework, .NET Core, or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using the `async` keyword are referred to as *async methods*.

The following example shows an `async` method. Almost everything in the code should look familiar to you.

You can find a complete Windows Presentation Foundation (WPF) example available for download from [Asynchronous programming with `async` and `await` in C#](#).

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

You can learn several practices from the preceding sample. Start with the method signature. It includes the `async` modifier. The return type is `Task<int>` (See "Return Types" section for more options). The method name ends in `Async`. In the body of the method, `GetStringAsync` returns a `Task<string>`. That means that when you `await` the task you'll get a `string` (`contents`). Before awaiting the task, you can do work that doesn't rely on the `string` from `GetStringAsync`.

Pay close attention to the `await` operator. It suspends `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` can't continue until `getStringTask` is complete.
- Meanwhile, control returns to the caller of `GetUrlContentLengthAsync`.
- Control resumes here when `getStringTask` is complete.
- The `await` operator then retrieves the `string` result from `getStringTask`.

The return statement specifies an integer result. Any methods that are awaiting `GetUrlContentLengthAsync` retrieve the length value.

If `GetUrlContentLengthAsync` doesn't have any work that it can do between calling `GetStringAsync` and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

```
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

The following characteristics summarize what makes the previous example an `async` method:

- The method signature includes an `async` modifier.

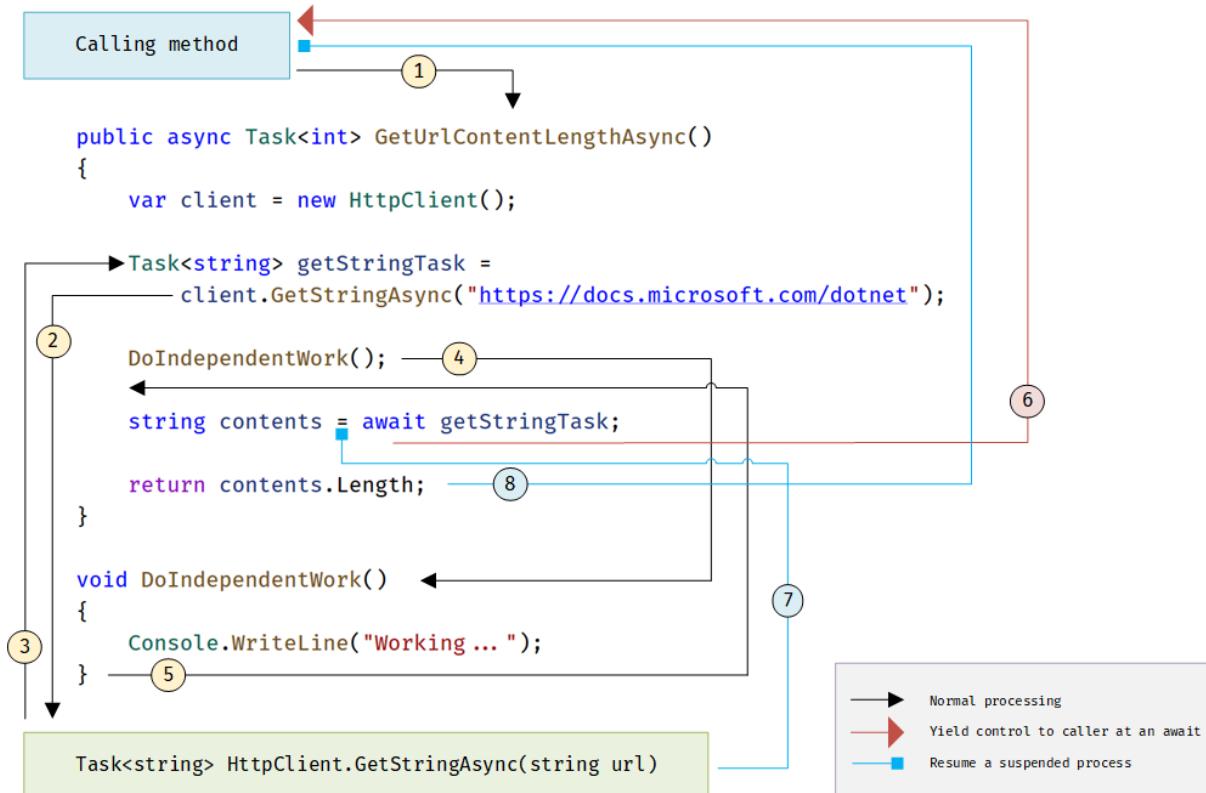
- The name of an async method, by convention, ends with an "Async" suffix.
 - The return type is one of the following types:
 - `Task<TResult>` if your method has a return statement in which the operand has type `TResult`.
 - `Task` if your method has no return statement or has a return statement with no operand.
 - `void` if you're writing an async event handler.
 - Any other type that has a `GetAwaiter` method (starting with C# 7.0).
- For more information, see the [Return types and parameters](#) section.
- The method usually includes at least one `await` expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

For more information about asynchrony in previous versions of .NET Framework, see [TPL and traditional .NET Framework asynchronous programming](#).

What happens in an async method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process:



2. `GetUrlContentLengthAsync` creates an `HttpClient` instance and calls the `GetStringAsync` asynchronous method to download the contents of a website as a string.
3. Something happens in `GetStringAsync` that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, `GetStringAsync` yields control to its caller, `GetUrlContentLengthAsync`.

`GetStringAsync` returns a `Task<TResult>`, where `TResult` is a string, and `GetUrlContentLengthAsync` assigns the task to the `getStringTask` variable. The task represents the ongoing process for the call to `GetStringAsync`, with a commitment to produce an actual string value when the work is complete.
4. Because `getStringTask` hasn't been awaited yet, `GetUrlContentLengthAsync` can continue with other work that doesn't depend on the final result from `GetStringAsync`. That work is represented by a call to the synchronous method `DoIndependentWork`.
5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.

6. `GetUrlContentLengthAsync` has run out of work that it can do without a result from `getStringTask`. `GetUrlContentLengthAsync` next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

Therefore, `GetUrlContentLengthAsync` uses an await operator to suspend its progress and to yield control to the method that called `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` returns a `Task<int>` to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

NOTE

If `GetStringAsync` (and therefore `getStringTask`) completes before `GetUrlContentLengthAsync` awaits it, control remains in `GetUrlContentLengthAsync`. The expense of suspending and then returning to `GetUrlContentLengthAsync` would be wasted if the called asynchronous process `getStringTask` has already completed and `GetUrlContentLengthAsync` doesn't have to wait for the final result.

- Inside the calling method the processing pattern continues. The caller might do other work that doesn't depend on the result from `GetUrlContentLengthAsync` before awaiting that result, or the caller might await immediately. The calling method is waiting for `GetUrlContentLengthAsync`, and `GetUrlContentLengthAsync` is waiting for `GetStringAsync`.
7. `GetStringAsync` completes and produces a string result. The string result isn't returned by the call to `GetStringAsync` in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, `getStringTask`. The await operator retrieves the result from `getStringTask`. The assignment statement assigns the retrieved result to `contents`.
 8. When `GetUrlContentLengthAsync` has the string result, the method can calculate the length of the string. Then the work of `GetUrlContentLengthAsync` is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result. If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

API async methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. .NET Framework 4.5 or higher and .NET Core contain many members that work with `async` and `await`. You can recognize them by the "Async" suffix that's appended to the member name, and by their return type of `Task` or `Task<TResult>`. For example, the `System.IO.Stream` class contains methods such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.

The Windows Runtime also contains many methods that you can use with `async` and `await` in Windows apps. For more information, see [Threading and async programming](#) for UWP development, and [Asynchronous programming \(Windows Store apps\)](#) and [Quickstart: Calling asynchronous APIs in C# or Visual Basic](#) if you use earlier versions of the Windows Runtime.

Threads

Async methods are intended to be non-blocking operations. An `await` expression in an async method doesn't block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The `async` and `await` keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use `Task.Run` to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

The `async`-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better than the `BackgroundWorker` class for I/O-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with the `Task.Run` method, `async` programming is better than `BackgroundWorker` for CPU-bound operations because `async` programming separates the coordination details of running your code from the work that `Task.Run` transfers to the thread pool.

async and await

If you specify that a method is an `async` method by using the `async` modifier, you enable the following two capabilities.

- The marked `async` method can use `await` to designate suspension points. The `await` operator tells the compiler that the `async` method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the `async` method.

The suspension of an `async` method at an `await` expression doesn't constitute an exit from the method, and `finally` blocks don't run.

- The marked `async` method can itself be awaited by methods that call it.

An `async` method typically contains one or more occurrences of an `await` operator, but the absence of `await` expressions doesn't cause a compiler error. If an `async` method doesn't use an `await` operator to mark a suspension point, the method executes as a synchronous method does, despite the `async` modifier. The compiler issues a warning for such methods.

`async` and `await` are contextual keywords. For more information and examples, see the following topics:

- [async](#)
- [await](#)

Return types and parameters

An async method typically returns a [Task](#) or a [Task<TResult>](#). Inside an async method, an `await` operator is applied to a task that's returned from a call to another async method.

You specify [Task<TResult>](#) as the return type if the method contains a `return` statement that specifies an operand of type `TResult`.

You use [Task](#) as the return type if the method has no return statement or has a return statement that doesn't return an operand.

Starting with C# 7.0, you can also specify any other return type, provided that the type includes a `GetAwaiter` method. [ValueTask<TResult>](#) is an example of such a type. It is available in the [System.Threading.Tasks.Extension](#) NuGet package.

The following example shows how you declare and call a method that returns a [Task<TResult>](#) or a [Task](#):

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}

Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also have a `void` return type. This return type is used primarily to define event handlers, where a `void` return type is required. Async event handlers often serve as the starting point for async programs.

An async method that has a `void` return type can't be awaited, and the caller of a void-returning method can't catch any exceptions that the method throws.

An async method can't declare `in`, `ref` or `out` parameters, but the method can call methods that have such parameters. Similarly, an async method can't return a value by reference, although it can call methods with `ref` return values.

For more information and examples, see [Async return types \(C#\)](#). For more information about how to catch exceptions in async methods, see [try-catch](#).

Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:

- [IAsyncOperation<TResult>](#), which corresponds to [Task<TResult>](#)
- [IAsyncAction](#), which corresponds to [Task](#)
- [IAsyncActionWithProgress<TProgress>](#)
- [IAsyncOperationWithProgress<TResult,TProgress>](#)

Naming convention

By convention, methods that return commonly awaitable types (for example, `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`) should have names that end with "Async". Methods that start an asynchronous operation but do not return an awaitable type should not have names that end with "Async", but may start with "Begin", "Start", or some other verb to suggest this method does not return or throw the result of the operation.

You can ignore the convention where an event, base class, or interface contract suggests a different name. For example, you shouldn't rename common event handlers, such as `OnButtonClick`.

Related topics and samples (Visual Studio)

TITLE	DESCRIPTION	SAMPLE
How to make multiple web requests in parallel by using async and await (C#)	Demonstrates how to start several tasks at the same time.	Async Sample: Make Multiple Web Requests in Parallel
Async return types (C#)	Illustrates the types that <code>async</code> methods can return, and explains when each type is appropriate.	
Cancel tasks with a cancellation token as a signaling mechanism.	Shows how to add the following functionality to your <code>async</code> solution: <ul style="list-style-type: none"> - Cancel a list of tasks (C#) - Cancel tasks after a period of time (C#) - Process asynchronous task as they complete (C#) 	
Using async for file access (C#)	Lists and demonstrates the benefits of using <code>async</code> and <code>await</code> to access files.	
Task-based asynchronous pattern (TAP)	Describes an asynchronous pattern, the pattern is based on the <code>Task</code> and <code>Task<TResult></code> types.	
Async Videos on Channel 9	Provides links to a variety of videos about <code>async</code> programming.	

See also

- [async](#)
- [await](#)
- [Asynchronous programming](#)
- [Async overview](#)

Async return types (C#)

3/20/2021 • 9 minutes to read • [Edit Online](#)

Async methods can have the following return types:

- [Task](#), for an async method that performs an operation but returns no value.
- [Task<TResult>](#), for an async method that returns a value.
- [void](#), for an event handler.
- Starting with C# 7.0, any type that has an accessible [GetAwaiter](#) method. The object returned by the [GetAwaiter](#) method must implement the [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#) interface.
- Starting with C# 8.0, [IAsyncEnumerable<T>](#), for an async method that returns an *async stream*.

For more information about async methods, see [Asynchronous programming with async and await \(C#\)](#).

Several other types also exist that are specific to Windows workloads:

- [DispatcherOperation](#), for async operations limited to Windows.
- [IAsyncAction](#), for async actions in UWP that do not return a value.
- [IAsyncActionWithProgress<TProgress>](#), for async actions in UWP that report progress but do not return a value.
- [IAsyncOperation<TResult>](#), for async operations in UWP that return a value.
- [IAsyncOperationWithProgress<TResult,TProgress>](#), for async operations in UWP that report progress and return a value.

Task return type

Async methods that don't contain a [return](#) statement or that contain a [return](#) statement that doesn't return an operand usually have a return type of [Task](#). Such methods return [void](#) if they run synchronously. If you use a [Task](#) return type for an async method, a calling method can use an [await](#) operator to suspend the caller's completion until the called async method has finished.

In the following example, the [WaitAndApologizeAsync](#) method doesn't contain a [return](#) statement, so the method returns a [Task](#) object. Returning a [Task](#) enables [WaitAndApologizeAsync](#) to be awaited. The [Task](#) type doesn't include a [Result](#) property because it has no return value.

```

public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//     Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.

```

`WaitAndApologizeAsync` is awaited by using an `await` statement instead of an `await` expression, similar to the calling statement for a synchronous void-returning method. The application of an `await` operator in this case doesn't produce a value. When the right operand of an `await` is a `Task<TResult>`, the `await` expression produces a result of `T`. When the right operand of an `await` is a `Task`, the `await` and its operand are a statement.

You can separate the call to `WaitAndApologizeAsync` from the application of an `await` operator, as the following code shows. However, remember that a `Task` doesn't have a `Result` property, and that no value is produced when an `await` operator is applied to a `Task`.

The following code separates calling the `WaitAndApologizeAsync` method from awaiting the task that the method returns.

```

Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);

```

Task<TResult> return type

The `Task<TResult>` return type is used for an `async` method that contains a `return` statement in which the operand is `TResult`.

In the following example, the `GetLeisureHoursAsync` method contains a `return` statement that returns an integer. Therefore, the method declaration must specify a return type of `Task<int>`. The `FromResult` `async` method is a placeholder for an operation that returns a `DayOfWeek`.

```

public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//   Today is Wednesday, May 24, 2017
//   Today's hours of leisure: 5

```

When `GetLeisureHoursAsync` is called from within an `await` expression in the `ShowTodaysInfo` method, the `await` expression retrieves the integer value (the value of `leisureHours`) that's stored in the task returned by the `GetLeisureHours` method. For more information about `await` expressions, see [await](#).

You can better understand how `await` retrieves the result from a `Task<T>` by separating the call to `GetLeisureHoursAsync` from the application of `await`, as the following code shows. A call to method `GetLeisureHoursAsync` that isn't immediately awaited returns a `Task<int>`, as you would expect from the declaration of the method. The task is assigned to the `getLeisureHoursTask` variable in the example. Because `getLeisureHoursTask` is a `Task<TResult>`, it contains a `Result` property of type `TResult`. In this case, `TResult` represents an integer type. When `await` is applied to `getLeisureHoursTask`, the `await` expression evaluates to the contents of the `Result` property of `getLeisureHoursTask`. The value is assigned to the `ret` variable.

IMPORTANT

The `Result` property is a blocking property. If you try to access it before its task is finished, the thread that's currently active is blocked until the task completes and the value is available. In most cases, you should access the value by using `await` instead of accessing the property directly.

The previous example retrieved the value of the `Result` property to block the main thread so that the `Main` method could print the `message` to the console before the application ended.

```

var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);

```

Void return type

You use the `void` return type in asynchronous event handlers, which require a `void` return type. For methods

other than event handlers that don't return a value, you should return a [Task](#) instead, because an async method that returns `void` can't be awaited. Any caller of such a method must continue to completion without waiting for the called async method to finish. The caller must be independent of any values or exceptions that the async method generates.

The caller of a void-returning async method can't catch exceptions thrown from the method, and such unhandled exceptions are likely to cause your application to fail. If a method that returns a [Task](#) or [Task<TResult>](#) throws an exception, the exception is stored in the returned task. The exception is rethrown when the task is awaited. Therefore, make sure that any async method that can produce an exception has a return type of [Task](#) or [Task<TResult>](#) and that calls to the method are awaited.

For more information about how to catch exceptions in async methods, see the [Exceptions in async methods](#) section of the [try-catch](#) article.

The following example shows the behavior of an async event handler. In the example code, an async event handler must let the main thread know when it finishes. Then the main thread can wait for an async event handler to complete before exiting the program.

```
using System;
using System.Threading.Tasks;

public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("    Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("    Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender, EventArgs e)
    {
        Console.WriteLine("    Handler 2 is starting...");
        Task.Delay(100).Wait();
    }
}
```

```

    task.Delay(100).Wait();
    Console.WriteLine("  Handler 2 is about to go async...");
    await Task.Delay(500);
    Console.WriteLine("  Handler 2 is done.");
    s_tcs.SetResult(true);
}

private static void OnButtonClicked3(object? sender, EventArgs e)
{
    Console.WriteLine("  Handler 3 is starting...");
    Task.Delay(100).Wait();
    Console.WriteLine("  Handler 3 is done.");
}
}

// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//   Handler 1 is starting...
//   Handler 1 is done.
//   Handler 2 is starting...
//   Handler 2 is about to go async...
//   Handler 3 is starting...
//   Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//   Handler 2 is done.

```

Generalized async return types and ValueTask<TResult>

Starting with C# 7.0, an `async` method can return any type that has an accessible `GetAwaiter` method that returns an instance of an *awaiter type*. In addition, the type returned from the `GetAwaiter` method must have the [System.Runtime.CompilerServices.AsyncMethodBuilderAttribute](#) attribute. You can learn more in the feature spec for [Task like return types](#).

This feature is the complement to [awaitable expressions](#), which describes the requirements for the operand of `await`. Generalized async return types enable the compiler to generate `async` methods that return different types. Generalized async return types enabled performance improvements in the .NET libraries. Because `Task` and `Task<TResult>` are reference types, memory allocation in performance-critical paths, particularly when allocations occur in tight loops, can adversely affect performance. Support for generalized return types means that you can return a lightweight value type instead of a reference type to avoid additional memory allocations.

.NET provides the [System.Threading.Tasks.ValueTask<TResult>](#) structure as a lightweight implementation of a generalized task-returning value. To use the [System.Threading.Tasks.ValueTask<TResult>](#) type, you must add the [System.Threading.Tasks.Extensions](#) NuGet package to your project. The following example uses the `ValueTask<TResult>` structure to retrieve the value of two dice rolls.

```

using System;
using System.Threading.Tasks;

class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}

// Example output:
//   Shaking dice...
//   You rolled 8

```

Writing a generalized async return type is an advanced scenario, and is targeted for use in very specific environments. Consider using the `Task`, `Task<T>` and `ValueTask<T>` types instead, which cover most scenarios for asynchronous code.

Async streams with `IAsyncEnumerable<T>`

Starting with C# 8.0, an async method may return an *async stream*, represented by `IAsyncEnumerable<T>`. An async stream provides a way to enumerate items read from a stream when elements are generated in chunks with repeated asynchronous calls. The following example shows an async method that generates an async stream:

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
        Here is the second line of text.
        And there is one more for good measure.
        Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ', StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

The preceding example reads lines from a string asynchronously. Once each line is read, the code enumerates each word in the string. Callers would enumerate each word using the `await foreach` statement. The method awaits when it needs to asynchronously read the next line from the source string.

See also

- [FromResult](#)
- [Process asynchronous tasks as they complete](#)
- [Asynchronous programming with `async` and `await` \(C#\)](#)
- [`async`](#)
- [`await`](#)

Cancel a list of tasks (C#)

3/6/2021 • 4 minutes to read • [Edit Online](#)

You can cancel an async console application if you don't want to wait for it to finish. By following the example in this topic, you can add a cancellation to an application that downloads the contents of a list of websites. You can cancel many tasks by associating the [CancellationTokenSource](#) instance with each task. If you select the Enter key, you cancel all tasks that aren't yet complete.

This tutorial covers:

- Creating a .NET console application
- Writing an async application that supports cancellation
- Demonstrating signaling cancellation

Prerequisites

This tutorial requires the following:

- [.NET 5.0 or later SDK](#)
- Integrated development environment (IDE)
 - We recommend [Visual Studio](#), [Visual Studio Code](#), or [Visual Studio for Mac](#)

Create example application

Create a new .NET Core console application. You can create one by using the `dotnet new console` command or from [Visual Studio](#). Open the *Program.cs* file in your favorite code editor.

Replace using statements

Replace the existing using statements with these declarations:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Add fields

In the `Program` class definition, add these three fields:

```

static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};

```

The [CancellationTokenSource](#) is used to signal a requested cancellation to a [CancellationToken](#). The [HttpClient](#) exposes the ability to send HTTP requests and receive HTTP responses. The [s_urlList](#) holds all of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the [Main](#) method. Replace the existing method with the following:

```

static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }
    });

    Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
    s_cts.Cancel();
});

    Task sumPageSizesTask = SumPageSizesAsync();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}

```

The updated `Main` method is now considered an [Async main](#), which allows for an asynchronous entry point into the executable. It writes a few instructional messages to the console, then declares a `Task` instance named `cancelTask`, which will read console key strokes. If the Enter key is pressed, a call to `CancellationTokenSource.Cancel()` is made. This will signal cancellation. Next, the `sumPageSizesTask` variable is assigned from the `SumPageSizesAsync` method. Both tasks are then passed to `Task.WhenAny(Task[])`, which will continue when any of the two tasks have completed.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}
```

The method starts by instantiating and starting a `Stopwatch`. It then loops through each URL in the `s_urlList` and calls `ProcessUrlAsync`. With each iteration, the `s_cts.Token` is passed into the `ProcessUrlAsync` method and the code returns a `Task<TResult>`, where `TResult` is an integer:

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The `CancellationToken` instance is passed into the `HttpClient.GetAsync(String, CancellationToken)` and `HttpContent.ReadAsByteArrayAsync()` methods. The `token` is used to register for requested cancellation. The length is returned after the URL and length is written to the console.

Example application output

```
Application started.  
Press the ENTER key to cancel...  
  
https://docs.microsoft.com 37,357  
https://docs.microsoft.com/aspnet/core 85,589  
https://docs.microsoft.com/azure 398,939  
https://docs.microsoft.com/azure/devops 73,663  
https://docs.microsoft.com/dotnet 67,452  
https://docs.microsoft.com/dynamics365 48,582  
https://docs.microsoft.com/education 22,924  
  
ENTER key pressed: cancelling downloads.  
  
Application ending.
```

Complete example

The following code is the complete text of the *Program.cs* file for the example.

```
using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
using System.Net.Http;  
using System.Threading;  
using System.Threading.Tasks;  
  
class Program  
{  
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();  
  
    static readonly HttpClient s_client = new HttpClient  
    {  
        MaxResponseContentBufferSize = 1_000_000  
    };  
  
    static readonly IEnumerable<string> s_urlList = new string[]  
    {  
        "https://docs.microsoft.com",  
        "https://docs.microsoft.com/aspnet/core",  
        "https://docs.microsoft.com/azure",  
        "https://docs.microsoft.com/azure/devops",  
        "https://docs.microsoft.com/dotnet",  
        "https://docs.microsoft.com/dynamics365",  
        "https://docs.microsoft.com/education",  
        "https://docs.microsoft.com/enterprise-mobility-security",  
        "https://docs.microsoft.com/gaming",  
        "https://docs.microsoft.com/graph",  
        "https://docs.microsoft.com/microsoft-365",  
        "https://docs.microsoft.com/office",  
        "https://docs.microsoft.com/powershell",  
        "https://docs.microsoft.com/sql",  
        "https://docs.microsoft.com/surface",  
        "https://docs.microsoft.com/system-center",  
        "https://docs.microsoft.com/visualstudio",  
        "https://docs.microsoft.com/windows",  
        "https://docs.microsoft.com/xamarin"  
    };  
  
    static async Task Main()  
    {  
        Console.WriteLine("Application started.");  
        Console.WriteLine("Press the ENTER key to cancel...\n");  
    }  
}
```

```

Task cancelTask = Task.Run(() =>
{
    while (Console.ReadKey().Key != ConsoleKey.Enter)
    {
        Console.WriteLine("Press the ENTER key to cancel...");
    }

    Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
    s_cts.Cancel();
});

Task sumPageSizesTask = SumPageSizesAsync();

await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urllist)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

```

See also

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Asynchronous programming with async and await \(C#\)](#)

Next steps

[Cancel async tasks after a period of time \(C#\)](#)

Cancel async tasks after a period of time (C#)

3/6/2021 • 2 minutes to read • [Edit Online](#)

You can cancel an asynchronous operation after a period of time by using the [CancellationTokenSource.CancelAfter](#) method if you don't want to wait for the operation to finish. This method schedules the cancellation of any associated tasks that aren't complete within the period of time that's designated by the `CancelAfter` expression.

This example adds to the code that's developed in [Cancel a list of tasks \(C#\)](#) to download a list of websites and to display the length of the contents of each one.

This tutorial covers:

- Updating an existing .NET console application
- Scheduling a cancellation

Prerequisites

This tutorial requires the following:

- You're expected to have created an application in the [Cancel a list of tasks \(C#\)](#) tutorial
- [.NET 5.0 or later SDK](#)
- Integrated development environment (IDE)
 - [We recommend Visual Studio, Visual Studio Code, or Visual Studio for Mac](#)

Update application entry point

Replace the existing `Main` method with the following:

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

The updated `Main` method writes a few instructional messages to the console. Within the `try catch`, a call to [CancellationTokenSource.CancelAfter\(Int32\)](#) schedules a cancellation. This will signal cancellation after a period of time.

Next, the `SumPageSizesAsync` method is awaited. If processing all of the URLs occurs faster than the scheduled cancellation, the application ends. However, if the scheduled cancellation is triggered before all of the URLs are processed, a `TaskCanceledException` is thrown.

Example application output

```
Application started.

https://docs.microsoft.com 37,357
https://docs.microsoft.com/aspnet/core 85,589
https://docs.microsoft.com/azure 398,939
https://docs.microsoft.com/azure/devops 73,663

Tasks cancelled: timed out.

Application ending.
```

Complete example

The following code is the complete text of the `Program.cs` file for the example.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        ...
    }
}
```

```

try
{
    s_cts.CancelAfter(3500);

    await SumPageSizesAsync();
}
catch (TaskCanceledException)
{
    Console.WriteLine("\nTasks cancelled: timed out.\n");
}
finally
{
    s_cts.Dispose();
}

Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

```

See also

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Asynchronous programming with async and await \(C#\)](#)
- [Cancel a list of tasks \(C#\)](#)

Process asynchronous tasks as they complete (C#)

3/6/2021 • 4 minutes to read • [Edit Online](#)

By using [Task.WhenAny](#), you can start multiple tasks at the same time and process them one by one as they're completed rather than process them in the order in which they're started.

The following example uses a query to create a collection of tasks. Each task downloads the contents of a specified website. In each iteration of a while loop, an awaited call to [WhenAny](#) returns the task in the collection of tasks that finishes its download first. That task is removed from the collection and processed. The loop repeats until the collection contains no more tasks.

Create example application

Create a new .NET Core console application. You can create one by using the [dotnet new console](#) command or from [Visual Studio](#). Open the *Program.cs* file in your favorite code editor.

Replace using statements

Replace the existing using statements with these declarations:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

Add fields

In the `Program` class definition, add the following two fields:

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

The `HttpClient` exposes the ability to send HTTP requests and receive HTTP responses. The `s_urlList` holds all of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the `Main` method. Replace the existing method with the following:

```
static Task Main() => SumPageSizesAsync();
```

The updated `Main` method is now considered an [Async main](#), which allows for an asynchronous entry point into the executable. It is expressed a call to `SumPageSizesAsync`.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

```

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

```

The method starts by instantiating and starting a [Stopwatch](#). It then includes a query that, when executed, creates a collection of tasks. Each call to [ProcessUrlAsync](#) in the following code returns a [Task<TResult>](#), where [TResult](#) is an integer:

```

IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);

```

Due to [deferred execution](#) with the LINQ, you call [Enumerable.ToList](#) to start each task.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

The [while](#) loop performs the following steps for each task in the collection:

1. Awaits a call to [WhenAny](#) to identify the first task in the collection that has finished its download.

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. Removes that task from the collection.

```
downloadTasks.Remove(finishedTask);
```

3. Awaits [finishedTask](#), which is returned by a call to [ProcessUrlAsync](#). The [finishedTask](#) variable is a [Task<TResult>](#) where [TResult](#) is an integer. The task is already complete, but you await it to retrieve the length of the downloaded website, as the following example shows. If the task is faulted, [await](#) will throw the first child exception stored in the [AggregateException](#), unlike reading the [Task<TResult>.Result](#) property, which would throw the [AggregateException](#).

```
total += await finishedTask;
```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The length is returned after the URL and length is written to the console.

Run the program several times to verify that the downloaded lengths don't always appear in the same order.

Caution

You can use `WhenAny` in a loop, as described in the example, to solve problems that involve a small number of tasks. However, other approaches are more efficient if you have a large number of tasks to process. For more information and examples, see [Processing tasks as they complete](#).

Complete example

The following code is the complete text of the `Program.cs` file for the example.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };

        static readonly IEnumerable<string> s_urlList = new string[]
        {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
        };
    }
}
```

```

};

static Task Main() => SumPageSizesAsync();

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

// Example output:
// https://docs.microsoft.com/windows          25,513
// https://docs.microsoft.com/gaming          30,705
// https://docs.microsoft.com/dotnet          69,626
// https://docs.microsoft.com/dynamics365      50,756
// https://docs.microsoft.com/surface          35,519
// https://docs.microsoft.com                39,531
// https://docs.microsoft.com/azure/devops     75,837
// https://docs.microsoft.com/xamarin          60,284
// https://docs.microsoft.com/system-center     43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946
// https://docs.microsoft.com/microsoft-365      43,278
// https://docs.microsoft.com/visualstudio       31,414
// https://docs.microsoft.com/office            42,292
// https://docs.microsoft.com/azure              401,113
// https://docs.microsoft.com/graph             46,831
// https://docs.microsoft.com/education          25,098
// https://docs.microsoft.com/powershell         58,173
// https://docs.microsoft.com/aspnet/core        87,763
// https://docs.microsoft.com/sql                 53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725

```

See also

- [WhenAny](#)
- [Asynchronous programming with async and await \(C#\)](#)

Asynchronous file access (C#)

11/2/2020 • 5 minutes to read • [Edit Online](#)

You can use the `async` feature to access files. By using the `async` feature, you can call into asynchronous methods without using callbacks or splitting your code across multiple methods or lambda expressions. To make synchronous code asynchronous, you just call an asynchronous method instead of a synchronous method and add a few keywords to the code.

You might consider the following reasons for adding asynchrony to file access calls:

- Asynchrony makes UI applications more responsive because the UI thread that launches the operation can perform other work. If the UI thread must execute code that takes a long time (for example, more than 50 milliseconds), the UI may freeze until the I/O is complete and the UI thread can again process keyboard and mouse input and other events.
- Asynchrony improves the scalability of ASP.NET and other server-based applications by reducing the need for threads. If the application uses a dedicated thread per response and a thousand requests are being handled simultaneously, a thousand threads are needed. Asynchronous operations often don't need to use a thread during the wait. They use the existing I/O completion thread briefly at the end.
- The latency of a file access operation might be very low under current conditions, but the latency may greatly increase in the future. For example, a file may be moved to a server that's across the world.
- The added overhead of using the `Async` feature is small.
- Asynchronous tasks can easily be run in parallel.

Use appropriate classes

The simple examples in this topic demonstrate `File.WriteAllTextAsync` and `File.ReadAllTextAsync`. For finite control over the file I/O operations, use the `FileStream` class, which has an option that causes asynchronous I/O to occur at the operating system level. By using this option, you can avoid blocking a thread pool thread in many cases. To enable this option, you specify the `useAsync=true` or `options=FileOptions.Asynchronous` argument in the constructor call.

You can't use this option with `StreamReader` and `StreamWriter` if you open them directly by specifying a file path. However, you can use this option if you provide them a `Stream` that the `FileStream` class opened.

Asynchronous calls are faster in UI apps even if a thread pool thread is blocked, because the UI thread isn't blocked during the wait.

Write text

The following examples write text to a file. At each `await` statement, the method immediately exits. When the file I/O is complete, the method resumes at the statement that follows the `await` statement. The `async` modifier is in the definition of methods that use the `await` statement.

Simple example

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Finite control example

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

The original example has the statement `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, which is a contraction of the following two statements:

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

The first statement returns a task and causes file processing to start. The second statement with the `await` causes the method to immediately exit and return a different task. When the file processing later completes, execution returns to the statement that follows the `await`.

Read text

The following examples read text from a file.

Simple example

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Finite control example

The text is buffered and, in this case, placed into a [StringBuilder](#). Unlike in the previous example, the evaluation of the `await` produces a value. The [ReadAsync](#) method returns a [Task<Int32>](#), so the evaluation of the `await` produces an `Int32` value `numRead` after the operation completes. For more information, see [Async Return Types \(C#\)](#).

```

public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}

```

Parallel asynchronous I/O

The following examples demonstrate parallel processing by writing 10 text files.

Simple example

```

public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}

```

Finite control example

For each file, the [WriteAsync](#) method returns a task that is then added to a list of tasks. The `await Task.WhenAll(tasks);` statement exits the method and resumes within the method when file processing is complete for all of the tasks.

The example closes all [FileStream](#) instances in a `finally` block after the tasks are complete. If each [FileStream](#) was instead created in a `using` statement, the [FileStream](#) might be disposed of before the task was complete.

Any performance boost is almost entirely from the parallel processing and not the asynchronous processing. The advantages of asynchrony are that it doesn't tie up multiple threads, and that it doesn't tie up the user interface thread.

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

When using the [WriteAsync](#) and [ReadAsync](#) methods, you can specify a [CancellationToken](#), which you can use to cancel the operation mid-stream. For more information, see [Cancellation in managed threads](#).

See also

- [Asynchronous programming with async and await \(C#\)](#)
- [Async return types \(C#\)](#)

Attributes (C#)

3/6/2021 • 5 minutes to read • [Edit Online](#)

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see [Reflection \(C#\)](#).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see [Creating Custom Attributes \(C#\)](#).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see [Accessing Attributes by Using Reflection \(C#\)](#).

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets ([]) above the declaration of the entity to which it applies.

In this example, the [SerializableAttribute](#) attribute is used to apply a specific characteristic to a class:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute [DllImportAttribute](#) is declared like the following example:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

NOTE

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

Attribute targets

The *target* of an attribute is the entity which the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

TARGET VALUE	APPLIES TO
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module
<code>field</code>	Field in a class or a struct
<code>event</code>	Event
<code>method</code>	Method or <code>get</code> and <code>set</code> property accessors

TARGET VALUE	APPLIES TO
param	Method parameters or <code>set</code> property accessor parameters
property	Property
return	Return value of a method, property indexer, or <code>get</code> property accessor
type	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an [auto-implemented property](#).

The following example shows how to apply attributes to assemblies and modules. For more information, see [Common Attributes \(C#\)](#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

NOTE

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see [AttributeUsage \(C#\)](#).

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.

- Calling unmanaged code using the [DllImportAttribute](#) class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Related sections

For more information, see:

- [Creating Custom Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)
- [How to create a C/C++ union by using attributes \(C#\)](#)
- [Common Attributes \(C#\)](#)
- [Caller Information \(C#\)](#)

See also

- [C# Programming Guide](#)
- [Reflection \(C#\)](#)
- [Attributes](#)
- [Using Attributes in C#](#)

Creating Custom Attributes (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from [Attribute](#), which makes identifying attribute definitions in metadata fast and easy. Suppose you want to tag types with the name of the programmer who wrote the type. You might define a custom [Author](#) attribute class:

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct)  
]  
public class AuthorAttribute : System.Attribute  
{  
    private string name;  
    public double version;  
  
    public AuthorAttribute(string name)  
    {  
        this.name = name;  
        version = 1.0;  
    }  
}
```

The class name `AuthorAttribute` is the attribute's name, `Author`, plus the `Attribute` suffix. It is derived from `System.Attribute`, so it is a custom attribute class. The constructor's parameters are the custom attribute's positional parameters. In this example, `name` is a positional parameter. Any public read-write fields or properties are named parameters. In this case, `version` is the only named parameter. Note the use of the `AttributeUsage` attribute to make the `Author` attribute valid only on class and `struct` declarations.

You could use this new attribute as follows:

```
[Author("P. Ackerman", version = 1.1)]  
class SampleClass  
{  
    // P. Ackerman's code goes here...  
}
```

`AttributeUsage` has a named parameter, `AllowMultiple`, with which you can make a custom attribute single-use or multiuse. In the following code example, a multiuse attribute is created.

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct,  
                      AllowMultiple = true) // multiuse attribute  
]  
public class AuthorAttribute : System.Attribute
```

In the following code example, multiple attributes of the same type are applied to a class.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

See also

- [System.Reflection](#)
- [C# Programming Guide](#)
- [Writing Custom Attributes](#)
- [Reflection \(C#\)](#)
- [Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)
- [AttributeUsage \(C#\)](#)

Accessing Attributes by Using Reflection (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The fact that you can define custom attributes and place them in your source code would be of little value without some way of retrieving that information and acting on it. By using reflection, you can retrieve the information that was defined with custom attributes. The key method is `GetCustomAttributes`, which returns an array of objects that are the run-time equivalents of the source code attributes. This method has several overloaded versions. For more information, see [Attribute](#).

An attribute specification such as:

```
[Author("P. Ackerman", version = 1.1)]  
class SampleClass
```

is conceptually equivalent to this:

```
Author anonymousAuthorObject = new Author("P. Ackerman");  
anonymousAuthorObject.version = 1.1;
```

However, the code is not executed until `SampleClass` is queried for attributes. Calling `GetCustomAttributes` on `SampleClass` causes an `Author` object to be constructed and initialized as above. If the class has other attributes, other attribute objects are constructed similarly. `GetCustomAttributes` then returns the `Author` object and any other attribute objects in an array. You can then iterate over this array, determine what attributes were applied based on the type of each array element, and extract information from the attribute objects.

Example

Here is a complete example. A custom attribute is defined, applied to several entities, and retrieved via reflection.

```
// Multiuse attribute.  
[System.AttributeUsage(System.AttributeTargets.Class |  
    System.AttributeTargets.Struct,  
    AllowMultiple = true) // Multiuse attribute.  
]  
public class Author : System.Attribute  
{  
    string name;  
    public double version;  
  
    public Author(string name)  
    {  
        this.name = name;  
  
        // Default value.  
        version = 1.0;  
    }  
  
    public string GetName()  
    {  
        return name;  
    }  
}  
  
// Class with the Author attribute.  
[Author("P. Ackerman")]
```

```

public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("  {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}

/* Output:
   Author information for FirstClass
   P. Ackerman, version 1.00
   Author information for SecondClass
   Author information for ThirdClass
   R. Koch, version 2.00
   P. Ackerman, version 1.00
*/

```

See also

- [System.Reflection](#)
- [Attribute](#)
- [C# Programming Guide](#)
- [Retrieving Information Stored in Attributes](#)
- [Reflection \(C#\)](#)
- [Attributes \(C#\)](#)
- [Creating Custom Attributes \(C#\)](#)

How to create a C/C++ union by using attributes (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

By using attributes, you can customize how structs are laid out in memory. For example, you can create what is known as a union in C/C++ by using the `StructLayout(LayoutKind.Explicit)` and `FieldOffset` attributes.

Example

In this code segment, all of the fields of `TestUnion` start at the same location in memory.

```
// Add a using directive for System.Runtime.InteropServices.  
  
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]  
struct TestUnion  
{  
    [System.Runtime.InteropServices.FieldOffset(0)]  
    public int i;  
  
    [System.Runtime.InteropServices.FieldOffset(0)]  
    public double d;  
  
    [System.Runtime.InteropServices.FieldOffset(0)]  
    public char c;  
  
    [System.Runtime.InteropServices.FieldOffset(0)]  
    public byte b;  
}
```

Example

The following is another example where fields start at different explicitly set locations.

```
// Add a using directive for System.Runtime.InteropServices.  
  
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]  
struct TestExplicit  
{  
    [System.Runtime.InteropServices.FieldOffset(0)]  
    public long lg;  
  
    [System.Runtime.InteropServices.FieldOffset(0)]  
    public int i1;  
  
    [System.Runtime.InteropServices.FieldOffset(4)]  
    public int i2;  
  
    [System.Runtime.InteropServices.FieldOffset(8)]  
    public double d;  
  
    [System.Runtime.InteropServices.FieldOffset(12)]  
    public char c;  
  
    [System.Runtime.InteropServices.FieldOffset(14)]  
    public byte b;  
}
```

The two integer fields, `i1` and `i2`, share the same memory locations as `lg`. This sort of control over struct layout is useful when using platform invocation.

See also

- [System.Reflection](#)
- [Attribute](#)
- [C# Programming Guide](#)
- [Attributes](#)
- [Reflection \(C#\)](#)
- [Attributes \(C#\)](#)
- [Creating Custom Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)

Collections (C#)

11/2/2020 • 13 minutes to read • [Edit Online](#)

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly typed objects. For information about arrays, see [Arrays](#).

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the [System.Collections.Generic](#) namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

NOTE

For the examples in this topic, include `using` directives for the `System.Collections.Generic` and `System.Linq` namespaces.

In this topic

- [Using a Simple Collection](#)
- [Kinds of Collections](#)
 - [System.Collections.Generic Classes](#)
 - [System.Collections.Concurrent Classes](#)
 - [System.Collections Classes](#)
- [Implementing a Collection of Key/Value Pairs](#)
- [Using LINQ to Access a Collection](#)
- [Sorting a Collection](#)
- [Defining a Custom Collection](#)
- [Iterators](#)

Using a Simple Collection

The examples in this section use the generic `List<T>` class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a `foreach` statement.

```
// Create a list of strings.  
var salmons = new List<string>();  
salmons.Add("chinook");  
salmons.Add("coho");  
salmons.Add("pink");  
salmons.Add("sockeye");  
  
// Iterate through the list.  
foreach (var salmon in salmons)  
{  
    Console.Write(salmon + " ");  
}  
// Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see [Object and Collection Initializers](#).

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```
// Create a list of strings by using a  
// collection initializer.  
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };  
  
// Iterate through the list.  
foreach (var salmon in salmons)  
{  
    Console.Write(salmon + " ");  
}  
// Output: chinook coho pink sockeye
```

You can use a `for` statement instead of a `foreach` statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using `for` instead of `foreach`.

```
// Create a list of strings by using a  
// collection initializer.  
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };  
  
for (var index = 0; index < salmons.Count; index++)  
{  
    Console.Write(salmons[index] + " ");  
}  
// Output: chinook coho pink sockeye
```

The following example removes an element from the collection by specifying the object to remove.

```

// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.WriteLine(salmon + " ");
}
// Output: chinook pink sockeye

```

The following example removes elements from a generic list. Instead of a `foreach` statement, a `for` statement that iterates in descending order is used. This is because the `RemoveAt` method causes elements after a removed element to have a lower index value.

```

var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.WriteLine(number + " "));
// Output: 0 2 4 6 8

```

For the type of elements in the `List<T>`, you can also define your own class. In the following example, the `Galaxy` class that is used by the `List<T>` is defined in the code.

```

private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new Galaxy() { Name="Tadpole", MegaLightYears=400},
        new Galaxy() { Name="Pinwheel", MegaLightYears=25},
        new Galaxy() { Name="Milky Way", MegaLightYears=0},
        new Galaxy() { Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears);
    }

    // Output:
    // Tadpole 400
    // Pinwheel 25
    // Milky Way 0
    // Andromeda 3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}

```

Kinds of Collections

Many common collections are provided by .NET. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- [System.Collections.Generic](#) classes
- [System.Collections.Concurrent](#) classes
- [System.Collections](#) classes

System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the [System.Collections.Generic](#) namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the [System.Collections.Generic](#) namespace:

CLASS	DESCRIPTION
Dictionary< TKey, TValue >	Represents a collection of key/value pairs that are organized based on the key.
List< T >	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue< T >	Represents a first in, first out (FIFO) collection of objects.
SortedList< TKey, TValue >	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer< T > implementation.

CLASS	DESCRIPTION
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see [Commonly Used Collection Types](#), [Selecting a Collection Class](#), and [System.Collections.Generic](#).

System.Collections.Concurrent Classes

In .NET Framework 4 and later versions, the collections in the [System.Collections.Concurrent](#) namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the [System.Collections.Concurrent](#) namespace should be used instead of the corresponding types in the [System.Collections.Generic](#) and [System.Collections](#) namespaces whenever multiple threads are accessing the collection concurrently. For more information, see [Thread-Safe Collections](#) and [System.Collections.Concurrent](#).

Some classes included in the [System.Collections.Concurrent](#) namespace are [BlockingCollection<T>](#), [ConcurrentDictionary< TKey, TValue >](#), [ConcurrentQueue<T>](#), and [ConcurrentStack<T>](#).

System.Collections Classes

The classes in the [System.Collections](#) namespace do not store elements as specifically typed objects, but as objects of type [Object](#).

Whenever possible, you should use the generic collections in the [System.Collections.Generic](#) namespace or the [System.Collections.Concurrent](#) namespace instead of the legacy types in the [System.Collections](#) namespace.

The following table lists some of the frequently used classes in the [System.Collections](#) namespace:

CLASS	DESCRIPTION
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

The [System.Collections.Specialized](#) namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

Implementing a Collection of Key/Value Pairs

The [Dictionary< TKey, TValue >](#) generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the [Dictionary](#) class is implemented as a hash table.

The following example creates a [Dictionary](#) collection and iterates through the dictionary by using a [foreach](#) statement.

```

private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

To instead use a collection initializer to build the `Dictionary` collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

```

private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

```

The following example uses the `ContainsKey` method and the `Item[]` property of `Dictionary` to quickly find an

item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements[symbol]` in C#.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

The following example instead uses the [TryGetValue](#) method quickly find an item by key.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see [Getting Started with LINQ in C#](#).

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

```

private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                where theElement.AtomicNumber < 22
                orderby theElement.Name
                select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    // Calcium 20
    // Potassium 19
    // Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19} },
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20} },
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21} },
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22} }
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a `List<T>`. The `Car` class implements the `IComparable<T>` interface, which requires that the `CompareTo` method be implemented.

Each call to the `CompareTo` method makes a single comparison that is used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the `Sort` method of the `List<T>` causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```

private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20} },
        { new Car() { Name = "car2", Color = "red", Speed = 50} },
        { new Car() { Name = "car3", Color = "green", Speed = 10} },
        { new Car() { Name = "car4", Color = "blue", Speed = 50} },
    }
}

```

```

        ...
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.WriteLine(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    // blue 50 car4
    // blue 30 car5
    // blue 20 car1
    // green 50 car7
    // green 10 car3
    // red 60 car6
    // red 50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}

```

Defining a Custom Collection

You can define a collection by implementing the [IEnumerable<T>](#) or [IEnumerable](#) interface.

Although you can define a custom collection, it is usually better to instead use the collections that are included in

.NET, which are described in [Kinds of Collections](#) earlier in this article.

The following example defines a custom collection class named `AllColors`. This class implements the `IEnumerable` interface, which requires that the `GetEnumerator` method be implemented.

The `GetEnumerator` method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the `IEnumerator` interface, which requires that the `Current` property, `MoveNext` method, and `Reset` method be implemented.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.WriteLine(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
        }
    }
}
```

```

        _position = -1;
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}

```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a `yield return` statement to return each element of the collection one at a time.

You call an iterator by using a `foreach` statement. Each iteration of the `foreach` loop calls the iterator. When a `yield return` statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see [Iterators \(C#\)](#).

The following example uses an iterator method. The iterator method has a `yield return` statement that is inside a `for` loop. In the `ListEvenNumbers` method, each iteration of the `foreach` statement body creates a call to the iterator method, which proceeds to the next `yield return` statement.

```

private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.WriteLine(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}

```

See also

- [Object and Collection Initializers](#)
- [Programming Concepts \(C#\)](#)
- [Option Strict Statement](#)
- [LINQ to Objects \(C#\)](#)
- [Parallel LINQ \(PLINQ\)](#)
- [Collections and Data Structures](#)
- [Selecting a Collection Class](#)

- Comparisons and Sorts Within Collections
- When to Use Generic Collections

Covariance and Contravariance (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
// Assignment compatibility.  
string str = "test";  
// An object of a more derived type is assigned to an object of a less derived type.  
object obj = str;  
  
// Covariance.  
IEnumerable<string> strings = new List<string>();  
// An object that is instantiated with a more derived type argument  
// is assigned to an object instantiated with a less derived type argument.  
// Assignment compatibility is preserved.  
IEnumerable<object> objects = strings;  
  
// Contravariance.  
// Assume that the following method is in the class:  
// static void SetObject(object o) { }  
Action<object> actObject = SetObject;  
// An object that is instantiated with a less derived type argument  
// is assigned to an object instantiated with a more derived type argument.  
// Assignment compatibility is reversed.  
Action<string> actString = actObject;
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
object[] array = new String[10];  
// The following statement produces a run-time exception.  
// array[0] = 10;
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see [Variance in Delegates \(C#\)](#) and [Using Variance in Delegates \(C#\)](#).

The following code example shows covariance and contravariance support for method groups.

```

static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}

```

In .NET Framework 4 and later versions, C# supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see [Variance in Generic Interfaces \(C#\)](#) and [Variance in Delegates \(C#\)](#).

The following code example shows implicit reference conversion for generic interfaces.

```

IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;

```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. C# enables you to create your own variant interfaces and delegates. For more information, see [Creating Variant Generic Interfaces \(C#\)](#) and [Variance in Delegates \(C#\)](#).

Related Topics

TITLE	DESCRIPTION
Variance in Generic Interfaces (C#)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in .NET.
Creating Variant Generic Interfaces (C#)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (C#)	Shows how covariance and contravariance support in the <code>IEnumerable<T></code> and <code>IComparable<T></code> interfaces can help you reuse code.
Variance in Delegates (C#)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in .NET.
Using Variance in Delegates (C#)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (C#)	Shows how covariance and contravariance support in the <code>Func</code> and <code>Action</code> delegates can help you reuse code.

Variance in Generic Interfaces (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

.NET Framework 4 introduced variance support for several existing generic interfaces. Variance support enables implicit conversion of classes that implement these interfaces.

Starting with .NET Framework 4, the following interfaces are variant:

- `IEnumerable<T>` (T is covariant)
- `IEnumerator<T>` (T is covariant)
- `IQueryable<T>` (T is covariant)
- `IGrouping< TKey, TElement >` (`TKey` and `TElement` are covariant)
- `IComparer<T>` (T is contravariant)
- `IEqualityComparer<T>` (T is contravariant)
- `IComparable<T>` (T is contravariant)

Starting with .NET Framework 4.5, the following interfaces are variant:

- `IReadOnlyList<T>` (T is covariant)
- `IReadOnlyCollection<T>` (T is covariant)

Covariance permits a method to have a more derived return type than that defined by the generic type parameter of the interface. To illustrate the covariance feature, consider these generic interfaces:

`IEnumerable<Object>` and `IEnumerable<String>`. The `IEnumerable<String>` interface does not inherit the `IEnumerable<Object>` interface. However, the `String` type does inherit the `Object` type, and in some cases you may want to assign objects of these interfaces to each other. This is shown in the following code example.

```
IEnumerable<String> strings = new List<String>();  
IEnumerable<Object> objects = strings;
```

In earlier versions of .NET Framework, this code causes a compilation error in C# and, if `Option Strict` is on, in Visual Basic. But now you can use `strings` instead of `objects`, as shown in the previous example, because the `IEnumerable<T>` interface is covariant.

Contravariance permits a method to have argument types that are less derived than that specified by the generic parameter of the interface. To illustrate contravariance, assume that you have created a `BaseComparer` class to compare instances of the `BaseClass` class. The `BaseComparer` class implements the `IEqualityComparer<BaseClass>` interface. Because the `IEqualityComparer<T>` interface is now contravariant, you can use `BaseComparer` to compare instances of classes that inherit the `BaseClass` class. This is shown in the following code example.

```

// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}

```

For more examples, see [Using Variance in Interfaces for Generic Collections \(C#\)](#).

Variance in generic interfaces is supported for reference types only. Value types do not support variance. For example, `IEnumerable<int>` cannot be implicitly converted to `IEnumerable<object>`, because integers are represented by a value type.

```

IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;

```

It is also important to remember that classes that implement variant interfaces are still invariant. For example, although `List<T>` implements the covariant interface `IEnumerable<T>`, you cannot implicitly convert `List<String>` to `List<Object>`. This is illustrated in the following code example.

```

// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();

```

See also

- [Using Variance in Interfaces for Generic Collections \(C#\)](#)
- [Creating Variant Generic Interfaces \(C#\)](#)
- [Generic Interfaces](#)
- [Variance in Delegates \(C#\)](#)

Creating Variant Generic Interfaces (C#)

11/2/2020 • 4 minutes to read • [Edit Online](#)

You can declare generic type parameters in interfaces as covariant or contravariant. *Covariance* allows interface methods to have more derived return types than that defined by the generic type parameters. *Contravariance* allows interface methods to have argument types that are less derived than that specified by the generic parameters. A generic interface that has covariant or contravariant generic type parameters is called *variant*.

NOTE

.NET Framework 4 introduced variance support for several existing generic interfaces. For the list of the variant interfaces in .NET, see [Variance in Generic Interfaces \(C#\)](#).

Declaring Variant Generic Interfaces

You can declare variant generic interfaces by using the `in` and `out` keywords for generic type parameters.

IMPORTANT

`ref`, `in`, and `out` parameters in C# cannot be variant. Value types also do not support variance.

You can declare a generic type parameter covariant by using the `out` keyword. The covariant type must satisfy the following conditions:

- The type is used only as a return type of interface methods and not used as a type of method arguments. This is illustrated in the following example, in which the type `R` is declared covariant.

```
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
    // void SetSomething(R sampleArg);

}
```

There is one exception to this rule. If you have a contravariant generic delegate as a method parameter, you can use the type as a generic type parameter for the delegate. This is illustrated by the type `R` in the following example. For more information, see [Variance in Delegates \(C#\)](#) and [Using Variance for Func and Action Generic Delegates \(C#\)](#).

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- The type is not used as a generic constraint for the interface methods. This is illustrated in the following code.

```

interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}

```

You can declare a generic type parameter contravariant by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a return type of interface methods. The contravariant type can also be used for generic constraints. The following code shows how to declare a contravariant interface and use a generic constraint for one of its methods.

```

interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}

```

It is also possible to support both covariance and contravariance in the same interface, but for different type parameters, as shown in the following code example.

```

interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}

```

Implementing Variant Generic Interfaces

You implement variant generic interfaces in classes by using the same syntax that is used for invariant interfaces. The following code example shows how to implement a covariant interface in a generic class.

```

interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}

```

Classes that implement variant interfaces are invariant. For example, consider the following code.

```

// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;

```

Extending Variant Generic Interfaces

When you extend a variant generic interface, you have to use the `in` and `out` keywords to explicitly specify whether the derived interface supports variance. The compiler does not infer the variance from the interface that is being extended. For example, consider the following interfaces.

```

interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }

```

In the `IInvariant<T>` interface, the generic type parameter `T` is invariant, whereas in `IExtCovariant<out T>` the type parameter is covariant, although both interfaces extend the same interface. The same rule is applied to contravariant generic type parameters.

You can create an interface that extends both the interface where the generic type parameter `T` is covariant and the interface where it is contravariant if in the extending interface the generic type parameter `T` is invariant. This is illustrated in the following code example.

```

interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }

```

However, if a generic type parameter `T` is declared covariant in one interface, you cannot declare it contravariant in the extending interface, or vice versa. This is illustrated in the following code example.

```

interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }

```

Avoiding Ambiguity

When you implement variant generic interfaces, variance can sometimes lead to ambiguity. Such ambiguity should be avoided.

For example, if you explicitly implement the same variant generic interface with different generic type parameters in one class, it can create ambiguity. The compiler does not produce an error in this case, but it's not specified which interface implementation will be chosen at run time. This ambiguity could lead to subtle bugs in your code. Consider the following code example.

```

// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because I Enumerable<out T> is covariant.
class Pets : I Enumerable<Cat>, I Enumerable<Dog>
{
    I Enumerator<Cat> I Enumerable<Cat>.Get Enumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    I Enumerator I Enumerable.Get Enumerator()
    {
        // Some code.
        return null;
    }

    I Enumerator<Dog> I Enumerable<Dog>.Get Enumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        I Enumerable<Animal> pets = new Pets();
        pets.Get Enumerator();
    }
}

```

In this example, it is unspecified how the `pets.Get Enumerator()` method chooses between `Cat` and `Dog`. This could cause problems in your code.

See also

- [Variance in Generic Interfaces \(C#\)](#)
- [Using Variance for Func and Action Generic Delegates \(C#\)](#)

Using Variance in Interfaces for Generic Collections (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A covariant interface allows its methods to return more derived types than those specified in the interface. A contravariant interface allows its methods to accept parameters of less derived types than those specified in the interface.

In .NET Framework 4, several existing interfaces became covariant and contravariant. These include `IEnumerable<T>` and `IComparable<T>`. This enables you to reuse methods that operate with generic collections of base types for collections of derived types.

For a list of variant interfaces in .NET, see [Variance in Generic Interfaces \(C#\)](#).

Converting Generic Collections

The following example illustrates the benefits of covariance support in the `IEnumerable<T>` interface. The `PrintFullName` method accepts a collection of the `IEnumerable<Person>` type as a parameter. However, you can reuse it for a collection of the `IEnumerable<Employee>` type because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
                person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);
    }
}
```

Comparing Generic Collections

The following example illustrates the benefits of contravariance support in the `IComparer<T>` interface. The `PersonComparer` class implements the `IComparer<Person>` interface. However, you can reuse this class to compare a sequence of objects of the `Employee` type because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
            new Employee() {FirstName = "Michael", LastName = "Alexander"},
            new Employee() {FirstName = "Jeff", LastName = "Price"}
        };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

See also

- [Variance in Generic Interfaces \(C#\)](#)

Variance in Delegates (C#)

11/2/2020 • 5 minutes to read • [Edit Online](#)

.NET Framework 3.5 introduced variance support for matching method signatures with delegate types in all delegates in C#. This means that you can assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. This includes both generic and non-generic delegates.

For example, consider the following code, which has two classes and two delegates: generic and non-generic.

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

When you create delegates of the `SampleDelegate` or `SampleGenericDelegate<A, R>` types, you can assign any one of the following methods to those delegates.

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFIRSTRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFIRSTRSecond(First first)
{ return new Second(); }
```

The following code example illustrates the implicit conversion between the method signature and the delegate type.

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFIRSTRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFIRSTRSecond;
```

For more examples, see [Using Variance in Delegates \(C#\)](#) and [Using Variance for Func and Action Generic Delegates \(C#\)](#).

Variance in Generic Type Parameters

In .NET Framework 4 or later you can enable implicit conversion between delegates, so that generic delegates that have different types specified by generic type parameters can be assigned to each other, if the types are inherited from each other as required by variance.

To enable implicit conversion, you must explicitly declare generic parameters in a delegate as covariant or contravariant by using the `in` or `out` keyword.

The following code example shows how you can create a delegate that has a covariant generic type parameter.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

If you use only variance support to match method signatures with delegate types and do not use the `in` and `out` keywords, you may find that sometimes you can instantiate delegates with identical lambda expressions or methods, but you cannot assign one delegate to another.

In the following code example, `SampleGenericDelegate<String>` cannot be explicitly converted to `SampleGenericDelegate<Object>`, although `String` inherits `Object`. You can fix this problem by marking the generic parameter `T` with the `out` keyword.

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Generic Delegates That Have Variant Type Parameters in .NET

.NET Framework 4 introduced variance support for generic type parameters in several existing generic delegates:

- `Action` delegates from the `System` namespace, for example, `Action<T>` and `Action<T1,T2>`
- `Func` delegates from the `System` namespace, for example, `Func<TResult>` and `Func<T,TResult>`

- The [Predicate<T>](#) delegate
- The [Comparison<T>](#) delegate
- The [Converter<TInput,TOutput>](#) delegate

For more information and examples, see [Using Variance for Func and Action Generic Delegates \(C#\)](#).

Declaring Variant Type Parameters in Generic Delegates

If a generic delegate has covariant or contravariant generic type parameters, it can be referred to as a *variant generic delegate*.

You can declare a generic type parameter covariant in a generic delegate by using the `out` keyword. The covariant type can be used only as a method return type and not as a type of method arguments. The following code example shows how to declare a covariant generic delegate.

```
public delegate R DCovariant<out R>();
```

You can declare a generic type parameter contravariant in a generic delegate by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a method return type. The following code example shows how to declare a contravariant generic delegate.

```
public delegate void DContravariant<in A>(A a);
```

IMPORTANT

`ref`, `in`, and `out` parameters in C# can't be marked as variant.

It is also possible to support both variance and covariance in the same delegate, but for different type parameters. This is shown in the following example.

```
public delegate R DVariant<in A, out R>(A a);
```

Instantiating and Invoking Variant Generic Delegates

You can instantiate and invoke variant delegates just as you instantiate and invoke invariant delegates. In the following example, the delegate is instantiated by a lambda expression.

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combining Variant Generic Delegates

Don't combine variant delegates. The [Combine](#) method does not support variant delegate conversion and expects delegates to be of exactly the same type. This can lead to a run-time exception when you combine delegates either by using the [Combine](#) method or by using the `+` operator, as shown in the following code example.

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Variance in Generic Type Parameters for Value and Reference Types

Variance for generic type parameters is supported for reference types only. For example, `DVariant<int>` can't be implicitly converted to `DVariant<Object>` or `DVariant<long>`, because integer is a value type.

The following example demonstrates that variance in generic type parameters is not supported for value types.

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

See also

- [Generics](#)
- [Using Variance for Func and Action Generic Delegates \(C#\)](#)
- [How to combine delegates \(Multicast Delegates\)](#)

Using Variance in Delegates (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

When you assign a method to a delegate, *covariance* and *contravariance* provide flexibility for matching a delegate type with a method signature. Covariance permits a method to have return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Example 1: Covariance

Description

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by `DogsHandler` is of type `Dogs`, which derives from the `Mammals` type that is defined in the delegate.

Code

```
class Mammals {}  
class Dogs : Mammals {}  
  
class Program  
{  
    // Define the delegate.  
    public delegate Mammals HandlerMethod();  
  
    public static Mammals MammalsHandler()  
    {  
        return null;  
    }  
  
    public static Dogs DogsHandler()  
    {  
        return null;  
    }  
  
    static void Test()  
    {  
        HandlerMethod handlerMammals = MammalsHandler;  
  
        // Covariance enables this assignment.  
        HandlerMethod handlerDogs = DogsHandler;  
    }  
}
```

Example 2: Contravariance

Description

This example demonstrates how delegates can be used with methods that have parameters whose types are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. The following example makes use of two delegates:

- A `KeyEventHandler` delegate that defines the signature of the `Button.KeyDown` event. Its signature is:

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- A [MouseEventHandler](#) delegate that defines the signature of the [Button.MouseClick](#) event. Its signature is:

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

The example defines an event handler with an [EventArgs](#) parameter and uses it to handle both the [Button.KeyDown](#) and [Button.MouseClick](#) events. It can do this because [EventArgs](#) is a base type of both [KeyEventArgs](#) and [MouseEventArgs](#).

Code

```
// Event handler that accepts a parameter of the EventArgs type.  
private void MultiHandler(object sender, System.EventArgs e)  
{  
    label1.Text = System.DateTime.Now.ToString();  
}  
  
public Form1()  
{  
    InitializeComponent();  
  
    // You can use a method that has an EventArgs parameter,  
    // although the event expects the KeyEventArgs parameter.  
    this.button1.KeyDown += this.MultiHandler;  
  
    // You can use the same method  
    // for an event that expects the MouseEventArgs parameter.  
    this.button1.MouseClick += this.MultiHandler;  
}
```

See also

- [Variance in Delegates \(C#\)](#)
- [Using Variance for Func and Action Generic Delegates \(C#\)](#)

Using Variance for Func and Action Generic Delegates (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

These examples demonstrate how to use covariance and contravariance in the `Func` and `Action` generic delegates to enable reuse of methods and provide more flexibility in your code.

For more information about covariance and contravariance, see [Variance in Delegates \(C#\)](#).

Using Delegates with Covariant Type Parameters

The following example illustrates the benefits of covariance support in the generic `Func` delegates. The `FindByTitle` method takes a parameter of the `String` type and returns an object of the `Employee` type. However, you can assign this method to the `Func<String, Person>` delegate because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Using Delegates with Contravariant Type Parameters

The following example illustrates the benefits of contravariance support in the generic `Action` delegates. The `AddToContacts` method takes a parameter of the `Person` type. However, you can assign this method to the `Action<Employee>` delegate because `Employee` inherits `Person`.

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

See also

- [Covariance and Contravariance \(C#\)](#)
- [Generics](#)

Expression Trees (C#)

11/2/2020 • 4 minutes to read • [Edit Online](#)

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to use expression trees to build dynamic queries \(C#\)](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the [System.Linq.Expressions](#) namespace.

Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type [Expression<TDelegate>](#), the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Creating Expression Trees by Using the API

To create expression trees by using the API, use the [Expression](#) class. This class contains static factory methods that create expression tree nodes of specific types, for example, [ParameterExpression](#), which represents a variable or parameter, or [MethodCallExpression](#), which represents a method call. [ParameterExpression](#), [MethodCallExpression](#), and the other expression-specific types are also defined in the [System.Linq.Expressions](#) namespace. These types derive from the abstract type [Expression](#).

The following code example demonstrates how to create an expression tree that represents the lambda expression `num => num < 5` by using the API.

```
// Add the following using directive to your code file:  
// using System.Linq.Expressions;  
  
// Manually build the expression tree for  
// the lambda expression num => num < 5.  
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");  
ConstantExpression five = Expression.Constant(5, typeof(int));  
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);  
Expression<Func<int, bool>> lambda1 =  
    Expression.Lambda<Func<int, bool>>(  
        numLessThanFive,  
        new ParameterExpression[] { numParam });
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the C# compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```
// Creating a parameter expression.  
ParameterExpression value = Expression.Parameter(typeof(int), "value");  
  
// Creating an expression to hold a local variable.  
ParameterExpression result = Expression.Parameter(typeof(int), "result");  
  
// Creating a label to jump to from a loop.  
LabelTarget label = Expression.Label(typeof(int));  
  
// Creating a method body.  
BlockExpression block = Expression.Block(  
    // Adding a local variable.  
    new[] { result },  
    // Assigning a constant to a local variable: result = 1  
    Expression.Assign(result, Expression.Constant(1)),  
    // Adding a loop.  
    Expression.Loop(  
        // Adding a conditional block into the loop.  
        Expression.IfThenElse(  
            // Condition: value > 1  
            Expression.GreaterThan(value, Expression.Constant(1)),  
            // If true: result *= value --  
            Expression.MultiplyAssign(result,  
                Expression.PostDecrementAssign(value)),  
            // If false, exit the loop and go to the label.  
            Expression.Break(label, result)  
        ),  
        // Label to jump to.  
        label  
    )  
);  
  
// Compile and execute an expression tree.  
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);  
  
Console.WriteLine(factorial);  
// Prints 120.
```

For more information, see [Generating Dynamic Methods with Expression Trees in Visual Studio 2010](#), which also applies to later versions of Visual Studio.

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression

`num => num < 5` can be decomposed into its parts.

```
// Add the following using directive to your code file:  
// using System.Linq.Expressions;  
  
// Create an expression tree.  
Expression<Func<int, bool>> exprTree = num => num < 5;  
  
// Decompose the expression tree.  
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];  
BinaryExpression operation = (BinaryExpression)exprTree.Body;  
ParameterExpression left = (ParameterExpression)operation.Left;  
ConstantExpression right = (ConstantExpression)operation.Right;  
  
Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",  
    param.Name, left.Name, operation.NodeType, right.Value);  
  
// This code produces the following output:  
  
// Decomposed expression: num => num LessThan 5
```

Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see [How to modify expression trees \(C#\)](#).

Compiling Expression Trees

The `Expression<TDelegate>` type provides the `Compile` method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```
// Creating an expression tree.  
Expression<Func<int, bool>> expr = num => num < 5;  
  
// Compiling the expression tree into a delegate.  
Func<int, bool> result = expr.Compile();  
  
// Invoking the delegate and writing the result to the console.  
Console.WriteLine(result(4));  
  
// Prints True.  
  
// You can also use simplified syntax  
// to compile and run an expression tree.  
// The following line can replace two previous statements.  
Console.WriteLine(expr.Compile()(4));  
  
// Also prints True.
```

For more information, see [How to execute expression trees \(C#\)](#).

See also

- [System.Linq.Expressions](#)
- [How to execute expression trees \(C#\)](#)

- [How to modify expression trees \(C#\)](#)
- [Lambda Expressions](#)
- [Dynamic Language Runtime Overview](#)
- [Programming Concepts \(C#\)](#)

How to execute expression trees (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows you how to execute an expression tree. Executing an expression tree may return a value, or it may just perform an action such as calling a method.

Only expression trees that represent lambda expressions can be executed. Expression trees that represent lambda expressions are of type [LambdaExpression](#) or [Expression<TDelegate>](#). To execute these expression trees, call the [Compile](#) method to create an executable delegate, and then invoke the delegate.

NOTE

If the type of the delegate is not known, that is, the lambda expression is of type [LambdaExpression](#) and not [Expression<TDelegate>](#), you must call the [DynamicInvoke](#) method on the delegate instead of invoking it directly.

If an expression tree does not represent a lambda expression, you can create a new lambda expression that has the original expression tree as its body, by calling the [Lambda<TDelegate>\(Expression, IEnumerable<ParameterExpression>\)](#) method. Then, you can execute the lambda expression as described earlier in this section.

Example

The following code example demonstrates how to execute an expression tree that represents raising a number to a power by creating a lambda expression and executing it. The result, which represents the number raised to the power, is displayed.

```
// The expression tree to execute.  
BinaryExpression be = Expression.Power(Expression.Constant(2D), Expression.Constant(3D));  
  
// Create a lambda expression.  
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);  
  
// Compile the lambda expression.  
Func<double> compiledExpression = le.Compile();  
  
// Execute the lambda expression.  
double result = compiledExpression();  
  
// Display the result.  
Console.WriteLine(result);  
  
// This code produces the following output:  
// 8
```

Compiling the Code

- Include the System.Linq.Expressions namespace.

See also

- [Expression Trees \(C#\)](#)
- [How to modify expression trees \(C#\)](#)

How to modify expression trees (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows you how to modify an expression tree. Expression trees are immutable, which means that they cannot be modified directly. To change an expression tree, you must create a copy of an existing expression tree and when you create the copy, make the required changes. You can use the [ExpressionVisitor](#) class to traverse an existing expression tree and to copy each node that it visits.

To modify an expression tree

1. Create a new **Console Application** project.
2. Add a `using` directive to the file for the `System.Linq.Expressions` namespace.
3. Add the `AndAlsoModifier` class to your project.

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right, b.IsLiftedToNull,
                b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

This class inherits the [ExpressionVisitor](#) class and is specialized to modify expressions that represent conditional `AND` operations. It changes these operations from a conditional `AND` to a conditional `OR`. To do this, the class overrides the [VisitBinary](#) method of the base type, because conditional `AND` expressions are represented as binary expressions. In the `VisitBinary` method, if the expression that is passed to it represents a conditional `AND` operation, the code constructs a new expression that contains the conditional `OR` operator instead of the conditional `AND` operator. If the expression that is passed to `VisitBinary` does not represent a conditional `AND` operation, the method defers to the base class implementation. The base class methods construct nodes that are like the expression trees that are passed in, but the nodes have their sub trees replaced with the expression trees that are produced recursively by the visitor.

4. Add a `using` directive to the file for the `System.Linq.Expressions` namespace.
5. Add code to the `Main` method in the `Program.cs` file to create an expression tree and pass it to the method that will modify it.

```
Expression<Func<string, bool>> expr = name => name.Length > 10 && name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression) expr);

Console.WriteLine(modifiedExpr);

/* This code produces the following output:

name => ((name.Length > 10) && name.StartsWith("G"))
name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

The code creates an expression that contains a conditional `AND` operation. It then creates an instance of the `AndAlsoModifier` class and passes the expression to the `Modify` method of this class. Both the original and the modified expression trees are outputted to show the change.

6. Compile and run the application.

See also

- [How to execute expression trees \(C#\)](#)
- [Expression Trees \(C#\)](#)

Querying based on runtime state (C#)

3/23/2021 • 8 minutes to read • [Edit Online](#)

Consider code that defines an [IQueryable](#) or an [IQueryable<T>](#) against a data source:

```
var companyNames = new[] {
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
};

// We're using an in-memory array as the data source, but the IQueryable could have come
// from anywhere -- an ORM backed by a database, a web request, or any other LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Every time you run this code, the same exact query will be executed. This is frequently not very useful, as you may want your code to execute different queries depending on conditions at runtime. This article describes how you can execute a different query based on runtime state.

IQueryable / IQueryable<T> and expression trees

Fundamentally, an [IQueryable](#) has two components:

- [Expression](#)—a language- and datasource-agnostic representation of the current query's components, in the form of an expression tree.
- [Provider](#)—an instance of a LINQ provider, which knows how to materialize the current query into a value or set of values.

In the context of dynamic querying, the provider will usually remain the same; the expression tree of the query will differ from query to query.

Expression trees are immutable; if you want a different expression tree—and thus a different query—you'll need to translate the existing expression tree to a new one, and thus to a new [IQueryable](#).

The following sections describe specific techniques for querying differently in response to runtime state:

- Use runtime state from within the expression tree
- Call additional LINQ methods
- Vary the expression tree passed into the LINQ methods
- Construct an [Expression<TDelegate>](#) expression tree using the factory methods at [Expression](#)
- Add method call nodes to an [IQueryable](#)'s expression tree
- Construct strings, and use the [Dynamic LINQ library](#)

Use runtime state from within the expression tree

Assuming the LINQ provider supports it, the simplest way to query dynamically is to reference the runtime state directly in the query via a closed-over variable, such as `length` in the following code example:

```

var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(", ", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(", ", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo

```

The internal expression tree—and thus the query—haven't been modified; the query returns different values only because the value of `length` has been changed.

Call additional LINQ methods

Generally, the [built-in LINQ methods](#) at [Queryable](#) perform two steps:

- Wrap the current expression tree in a [MethodCallExpression](#) representing the method call.
- Pass the wrapped expression tree back to the provider, either to return a value via the provider's [IQueryProvider.Execute](#) method; or to return a translated query object via the [IQueryProvider.CreateQuery](#) method.

You can replace the original query with the result of an [IQueryable<T>](#)-returning method, to get a new query. You can do this conditionally based on runtime state, as in the following example:

```

// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}

```

Vary the expression tree passed into the LINQ methods

You can pass in different expressions to the LINQ methods, depending on runtime state:

```

// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("", "") => x => true,
    (_, "") => x => x.StartsWith(startsWith),
    ("", _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);

```

You might also want to compose the various sub-expressions using a third-party library such as [LinqKit's PredicateBuilder](#):

```

// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);

```

Construct expression trees and queries using factory methods

In all the examples up to this point, we've known the element type at compile time—`string`—and thus the type of the query—`IQueryable<string>`. You may need to add components to a query of any element type, or to add different components, depending on the element type. You can create expression trees from the ground up, using the factory methods at [System.Linq.Expressions.Expression](#), and thus tailor the expression at runtime to a specific element type.

Constructing an `Expression<TDelegate>`

When you construct an expression to pass into one of the LINQ methods, you're actually constructing an instance of `Expression<TDelegate>`, where `TDelegate` is some delegate type such as `Func<string, bool>`, `Action`, or a custom delegate type.

`Expression<TDelegate>` inherits from [LambdaExpression](#), which represents a complete lambda expression like the following:

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

A [LambdaExpression](#) has two components:

- a parameter list—`(string x)`—represented by the `Parameters` property
- a body—`x.StartsWith("a")`—represented by the `Body` property.

The basic steps in constructing an `Expression<TDelegate>` are as follows:

- Define [ParameterExpression](#) objects for each of the parameters (if any) in the lambda expression, using the `Parameter` factory method.

```
ParameterExpression x = Parameter(typeof(string), "x");
```

- Construct the body of your [LambdaExpression](#), using the `ParameterExpression`(s) you've defined, and the factory methods at `Expression`. For instance, an expression representing `x.StartsWith("a")` could be constructed like this:

```
Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", new[] { typeof(string) })!,
    Constant("a")
);
```

- Wrap the parameters and body in a compile-time-typed [Expression<TDelegate>](#), using the appropriate [Lambda](#) factory method overload:

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body, x);
```

The following sections describe a scenario in which you might want to construct an [Expression<TDelegate>](#) to pass into a LINQ method, and provide a complete example of how to do so using the factory methods.

Scenario

Let's say you have multiple entity types:

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

For any of these entity types, you want to filter and return only those entities that have a given text inside one of their `string` fields. For `Person`, you'd want to search the `FirstName` and `LastName` properties:

```
string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

but for `Car`, you'd want to search only the `Model` property:

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

While you could write one custom function for `IQueryable<Person>` and another for `IQueryable<Car>`, the following function adds this filtering to any existing query, irrespective of the specific element type.

Example

```

// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains", new[] { typeof(string) })!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree node like
            x.PropertyName.Contains("term")
                Call(
                    Property(          // .Contains(...)
                        prm,           // .PropertyName
                        prp           // x
                    ),
                    containsMethod,
                    Constant(term)   // "term"
                )
        );
}

// Combine all the resultant expression nodes using ||
Expression body = expressions
    .Aggregate(
        (prev, current) => Or(prev, current)
    );

// Wrap the expression body in a compile-time-typed lambda expression
Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

// Because the lambda is compile-time-typed (albeit with a generic parameter), we can use it with the
// Where method
return source.Where(lambda);
}

```

Because the `TextFilter` function takes and returns an `IQueryable<T>` (and not just an `IQueryable`), you can add further compile-time-typed query elements after the text filter.

```

var qry = TextFilter(
    new List<Person>().AsQueryable(),
    "abcd"
)
.Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
    new List<Car>().AsQueryable(),
    "abcd"
)
.Where(x => x.Year == 2010);

```

Add method call nodes to the [IQueryable](#)'s expression tree

If you have an [IQueryable](#) instead of an [IQueryable<T>](#), you can't directly call the generic LINQ methods. One alternative is to build the inner expression tree as above, and use reflection to invoke the appropriate LINQ method while passing in the expression tree.

You could also duplicate the LINQ method's functionality, by wrapping the entire tree in a [MethodCallExpression](#) which represents a call to the LINQ method:

```

IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the LambdaExpression's body is the same as in the
    previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) = constructBody(elementType, term);
    if (body is null) {return source;}

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        new[] { elementType },
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}

```

Note that in this case you don't have a compile-time `T` generic placeholder, so you'll use the [Lambda](#) overload which doesn't require compile-time type information, and which produces a [LambdaExpression](#) instead of an [Expression<TDelegate>](#).

The Dynamic LINQ library

Constructing expression trees using factory methods is relatively complex; it is easier to compose strings. The [Dynamic LINQ library](#) exposes a set of extension methods on [IQueryable](#) corresponding to the standard LINQ methods at [Queryable](#), and which accept strings in a [special syntax](#) instead of expression trees. The library generates the appropriate expression tree from the string, and can return the resultant translated [IQueryable](#).

For instance, the previous example could be rewritten as follows:

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter.Strings(IQueryable source, string term) {
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{{prp.Name}}.Contains(@0)")
    );

    return source.Where(filterExpr, term);
}
```

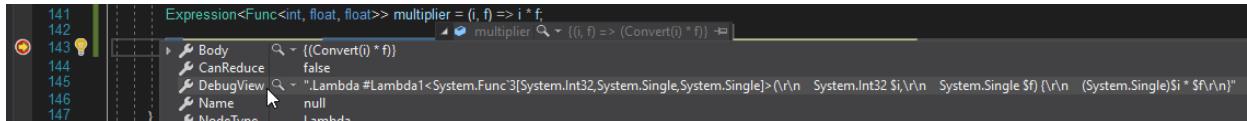
See also

- [Expression Trees \(C#\)](#)
- [How to execute expression trees \(C#\)](#)
- [Dynamically specify predicate filters at runtime](#)

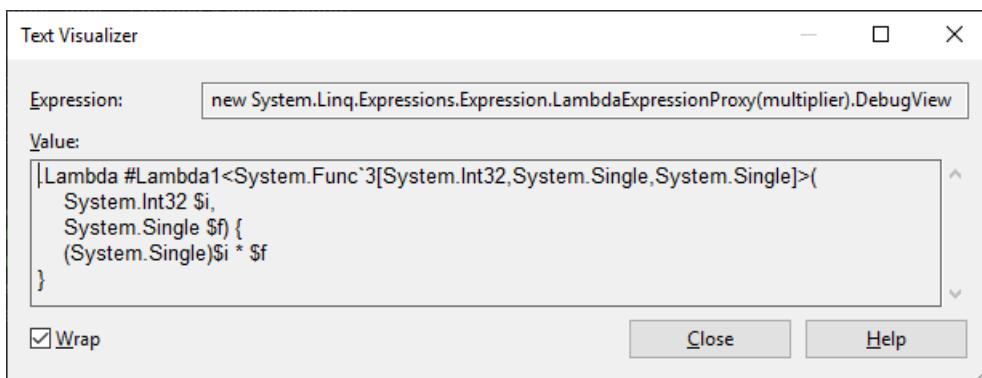
Debugging Expression Trees in Visual Studio (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can analyze the structure and content of expression trees when you debug your applications. To get a quick overview of the expression tree structure, you can use the `DebugView` property, which represents expression trees [using a special syntax](#). (Note that `DebugView` is available only in debug mode.)

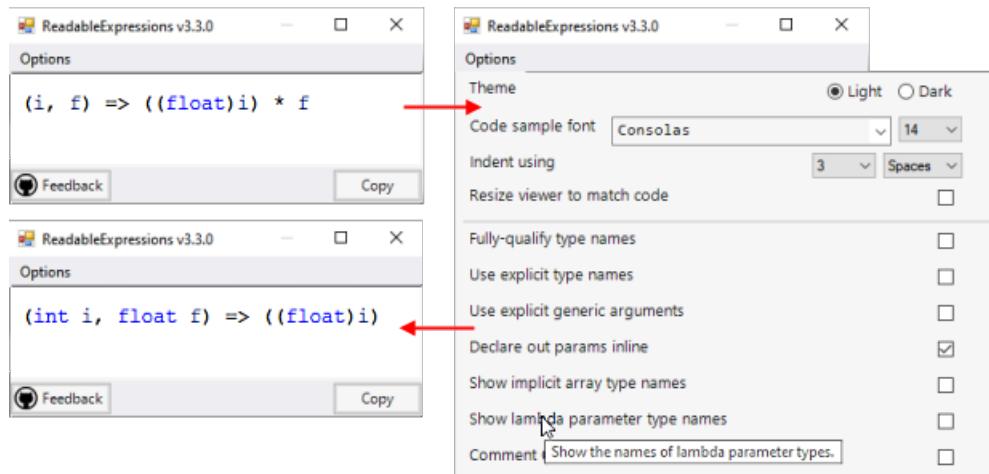


Since `DebugView` is a string, you can use the [built-in Text Visualizer](#) to view it across multiple lines, by selecting **Text Visualizer** from the magnifying glass icon next to the `DebugView` label.

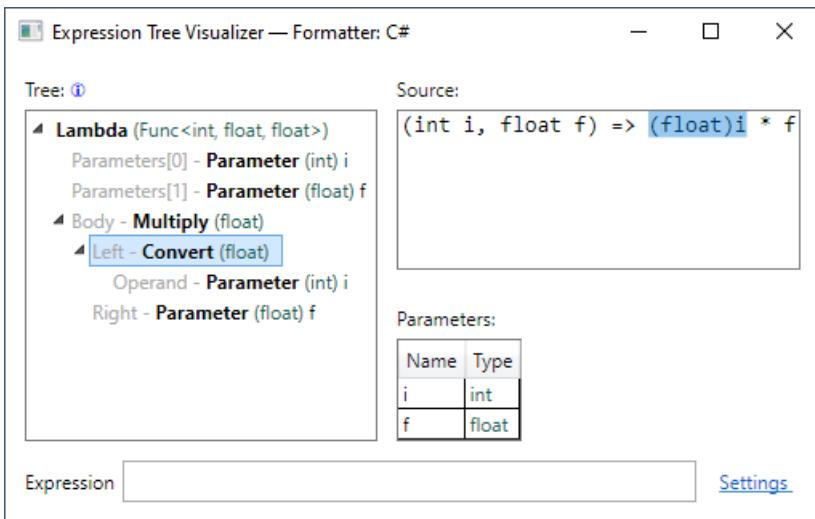


Alternatively, you can install and use a [custom visualizer](#) for expression trees, such as:

- [Readable Expressions \(MIT license\)](#), available at the [Visual Studio Marketplace](#), renders the expression tree as themeable C# code, with various rendering options:



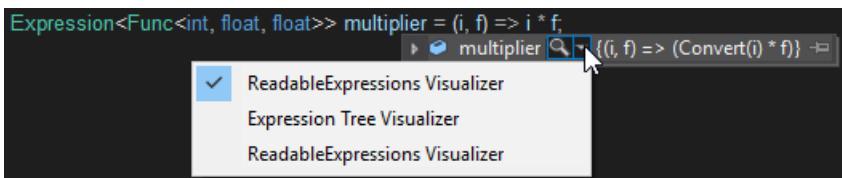
- [Expression Tree Visualizer \(MIT license\)](#) provides a tree view of the expression tree and its individual nodes:



To open a visualizer for an expression tree

1. Click the magnifying glass icon that appears next to the expression tree in **DataTips**, a **Watch** window, the **Autos** window, or the **Locals** window.

A list of available visualizers is displayed.:



2. Click the visualizer you want to use.

See also

- [Expression Trees \(C#\)](#)
- [Debugging in Visual Studio](#)
- [Create Custom Visualizers](#)
- [DebugView Syntax](#)

DebugView syntax

3/6/2021 • 2 minutes to read • [Edit Online](#)

The **DebugView** property (available only when debugging) provides a string rendering of expression trees. Most of the syntax is fairly straightforward to understand; the special cases are described in the following sections.

Each example is followed by a block comment, containing the **DebugView**.

ParameterExpression

ParameterExpression variable names are displayed with a `$` symbol at the beginning.

If a parameter does not have a name, it is assigned an automatically generated name, such as `$var1` or `$var2`.

Examples

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
/*
    $num
*/
ParameterExpression numParam = Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

For **ConstantExpression** objects that represent integer values, strings, and `null`, the value of the constant is displayed.

For numeric types that have standard suffixes as C# literals, the suffix is added to the value. The following table shows the suffixes associated with various numeric types.

TYPE	KEYWORD	SUFFIX
<code>System.UInt32</code>	<code>uint</code>	<code>U</code>
<code>System.Int64</code>	<code>long</code>	<code>L</code>
<code>System.UInt64</code>	<code>ulong</code>	<code>UL</code>
<code>System.Double</code>	<code>double</code>	<code>D</code>
<code>System.Single</code>	<code>float</code>	<code>F</code>
<code>System.Decimal</code>	<code>decimal</code>	<code>M</code>

Examples

```

int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/

```

BlockExpression

If the type of a [BlockExpression](#) object differs from the type of the last expression in the block, the type is displayed within angle brackets (`<` and `>`). Otherwise, the type of the [BlockExpression](#) object is not displayed.

Examples

```

BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block = Expression.Block(typeof(Object), Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/

```

LambdaExpression

[LambdaExpression](#) objects are displayed together with their delegate types.

If a lambda expression does not have a name, it is assigned an automatically generated name, such as `#Lambda1` or `#Lambda2`.

Examples

```

LambdaExpression lambda = Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda = Expression.Lambda<Func<int>>(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/

```

LabelExpression

If you specify a default value for the [LabelExpression](#) object, this value is displayed before the [LabelTarget](#) object.

The `.Label` token indicates the start of the label. The `.LabelTarget` token indicates the destination of the target to jump to.

If a label does not have a name, it is assigned an automatically generated name, such as `#Label1` or `#Label2`.

Examples

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
        -1
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Checked Operators

Checked operators are displayed with the `#` symbol in front of the operator. For example, the checked addition operator is displayed as `#+`.

Examples

```
Expression expr = Expression.AddChecked( Expression.Constant(1), Expression.Constant(2));
/*
    1 #+ 2
*/
Expression expr = Expression.ConvertChecked( Expression.Constant(10.0), typeof(int));
/*
    #(System.Int32)10D
*/
```

Iterators (C#)

11/2/2020 • 7 minutes to read • [Edit Online](#)

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or `get` accessor performs a custom iteration over a collection. An iterator method uses the `yield return` statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a `foreach` statement or by using a LINQ query.

In the following example, the first iteration of the `foreach` loop causes execution to proceed in the `SomeNumbers` iterator method until the first `yield return` statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a `yield return` statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.WriteLine(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

The return type of an iterator method or `get` accessor can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

You can use a `yield break` statement to end the iteration.

NOTE

For all examples in this topic except the Simple Iterator example, include `using` directives for the `System.Collections` and `System.Collections.Generic` namespaces.

Simple Iterator

The following example has a single `yield return` statement that is inside a `for` loop. In `Main`, each iteration of the `foreach` statement body creates a call to the iterator function, which proceeds to the next `yield return` statement.

```

static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.WriteLine(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}

```

Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the `IEnumerable` interface, which requires a `GetEnumerator` method. The compiler implicitly calls the `GetEnumerator` method, which returns an `IEnumerator`.

The `GetEnumerator` method returns each string one at a time by using the `yield return` statement.

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.WriteLine(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}

```

The following example creates a `Zoo` class that contains a collection of animals.

The `foreach` statement that refers to the class instance (`theZoo`) implicitly calls the `GetEnumerator` method. The `foreach` statements that refer to the `Birds` and `Mammals` properties use the `AnimalsForType` named iterator method.

```

static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
}

```

```

// ..private methods.

private IEnumerable AnimalsForType(Animal.TypeEnum type)
{
    foreach (Animal theAnimal in animals)
    {
        if (theAnimal.Type == type)
        {
            yield return theAnimal.Name;
        }
    }
}

// Private class.
private class Animal
{
    public enum TypeEnum { Bird, Mammal }

    public string Name { get; set; }
    public TypeEnum Type { get; set; }
}
}

```

Using Iterators with a Generic List

In the following example, the `Stack<T>` generic class implements the `IEnumerable<T>` generic interface. The `Push` method assigns values to an array of type `T`. The `GetEnumerator` method returns the array values by using the `yield return` statement.

In addition to the generic `GetEnumerator` method, the non-generic `GetEnumerator` method must also be implemented. This is because `IEnumerable<T>` inherits from `IEnumerable`. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property uses an iterator in a `get` accessor.

```

static void Main()
{
    Stack<int> theStack = new Stack<int>();

    // Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)

```

```

    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }

    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerable IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)

```

```

        for (int index = top - 1, index -= startIndex, index--)
        {
            yield return values[index];
        }
    }
}

```

Syntax Information

An iterator can occur as a method or `get` accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static finalizer.

An implicit conversion must exist from the expression type in the `yield return` statement to the type argument for the `IEnumerable<T>` returned by the iterator.

In C#, an iterator method cannot have any `in`, `ref`, or `out` parameters.

In C#, `yield` is not a reserved word and has special meaning only when it is used before a `return` or `break` keyword.

Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the `foreach` loop in the client code continues.

To see what the compiler does, you can use the `Ildasm.exe` tool to view the Microsoft intermediate language code that's generated for an iterator method.

When you create an iterator for a `class` or `struct`, you don't have to implement the whole `IEnumerator` interface. When the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the `IEnumerator` or `IEnumerator<T>` interface.

On each successive iteration of the `foreach` loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous `yield return` statement. It then continues to the next `yield return` statement until the end of the iterator body is reached, or until a `yield break` statement is encountered.

Iterators don't support the `IEnumerator.Reset` method. To reiterate from the start, you must obtain a new iterator. Calling `Reset` on the iterator returned by an iterator method throws a `NotSupportedException`.

For additional information, see the [C# Language Specification](#).

Use of Iterators

Iterators enable you to maintain the simplicity of a `foreach` loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first `foreach` loop iteration.
- Avoid fully loading a large list before the first iteration of a `foreach` loop. An example is a paged fetch to load a batch of table rows. Another example is the `EnumerateFiles` method, which implements iterators in .NET.
- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

See also

- [System.Collections.Generic](#)
- [IEnumerable<T>](#)
- [foreach, in](#)
- [yield](#)
- [Using foreach with Arrays](#)
- [Generics](#)

Language Integrated Query (LINQ)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events. You write queries against strongly typed collections of objects by using language keywords and familiar operators. The LINQ family of technologies provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML).

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO.NET Datasets, XML documents and streams, and .NET collections.

You can write LINQ queries in C# for SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports `IEnumerable` or the generic `IEnumerable<T>` interface. LINQ support is also provided by third parties for many Web services and other database implementations.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

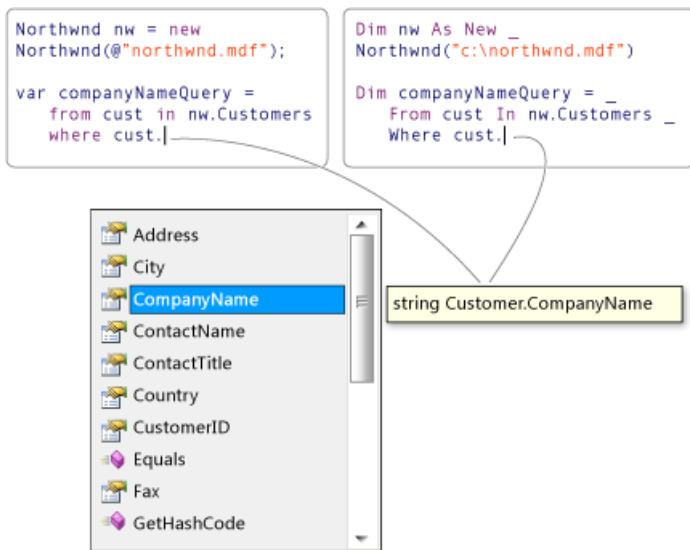
```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        I Enumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

The following illustration from Visual Studio shows a partially-completed LINQ query against a SQL Server database in both C# and Visual Basic with full type checking and IntelliSense support:



Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see [Type relationships in LINQ query operations](#).
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see [Introduction to LINQ queries](#).
- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see [C# language specification](#) and [Standard query operators overview](#).
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as `Count` or `Max`, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query syntax and method syntax in LINQ](#).
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. `IEnumerable<T>` queries are compiled to delegates. `IQueryable` and `IQueryable<T>` queries are compiled to expression trees. For more information, see [Expression trees](#).

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in [Query expression basics](#), and then read the documentation for the LINQ technology in which you are interested:

- XML documents: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to entities](#)
- .NET collections, files, strings and so on: [LINQ to objects](#)

To gain a deeper understanding of LINQ in general, see [LINQ in C#](#).

To start working with LINQ in C#, see the tutorial [Working with LINQ](#).

Introduction to LINQ Queries (C#)

11/2/2020 • 6 minutes to read • [Edit Online](#)

A *query* is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language. Different languages have been developed over time for the various types of data sources, for example SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they must support. LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

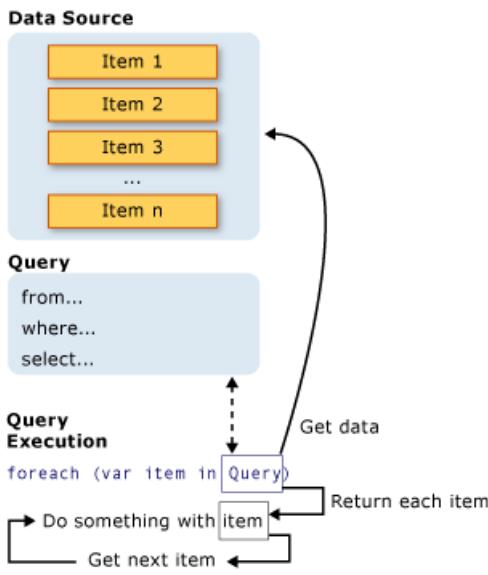
The following example shows how the three parts of a query operation are expressed in source code. The example uses an integer array as a data source for convenience; however, the same concepts apply to other data sources also. This example is referred to throughout the rest of this topic.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0}, ", num);
        }
    }
}
```

The following illustration shows the complete query operation. In LINQ, the execution of the query is distinct from the query itself. In other words, you have not retrieved any data just by creating a query variable.



The Data Source

In the previous example, because the data source is an array, it implicitly supports the generic `IEnumerable<T>` interface. This fact means it can be queried with LINQ. A query is executed in a `foreach` statement, and `foreach` requires `IEnumerable` or `IEnumerable<T>`. Types that support `IEnumerable<T>` or a derived interface such as the generic `IQueryable<T>` are called *queryable types*.

A queryable type requires no modification or special treatment to serve as a LINQ data source. If the source data is not already in memory as a queryable type, the LINQ provider must represent it as such. For example, LINQ to XML loads an XML document into a queryable `XElement` type:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
 XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

With LINQ to SQL, you first create an object-relational mapping at design time either manually or by using the [LINQ to SQL Tools in Visual Studio](#). You write your queries against the objects, and at run-time LINQ to SQL handles the communication with the database. In the following example, `Customers` represents a specific table in the database, and the type of the query result, `IQueryable<T>`, derives from `IEnumerable<T>`.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

For more information about how to create specific types of data sources, see the documentation for the various LINQ providers. However, the basic rule is very simple: a LINQ data source is any object that supports the generic `IEnumerable<T>` interface, or an interface that inherits from it.

NOTE

Types such as `ArrayList` that support the non-generic `IEnumerable` interface can also be used as a LINQ data source. For more information, see [How to query an ArrayList with LINQ \(C#\)](#).

The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. A query is stored in a query variable and initialized with a query expression. To make it easier to write queries, C# has introduced new query syntax.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: `from`, `where` and `select`. (If you are familiar with SQL, you will have noticed that the ordering of the clauses is reversed from the order in SQL.) The `from` clause specifies the data source, the `where` clause applies the filter, and the `select` clause specifies the type of the returned elements. These and the other query clauses are discussed in detail in the [Language Integrated Query \(LINQ\)](#) section. For now, the important point is that in LINQ, the query variable itself takes no action and returns no data. It just stores the information that is required to produce the results when the query is executed at some later point. For more information about how queries are constructed behind the scenes, see [Standard Query Operators Overview \(C#\)](#).

NOTE

Queries can also be expressed by using method syntax. For more information, see [Query Syntax and Method Syntax in LINQ](#).

Query Execution

Deferred Execution

As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a `foreach` statement. This concept is referred to as *deferred execution* and is demonstrated in the following example:

```
// Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0}, ", num);
}
```

The `foreach` statement is also where the query results are retrieved. For example, in the previous query, the iteration variable `num` holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it as often as you like. For example, you may have a database that is being updated continually by a separate application. In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time.

Forcing Immediate Execution

Queries that perform aggregation functions over a range of source elements must first iterate over those elements. Examples of such queries are `Count`, `Max`, `Average`, and `First`. These execute without an explicit `foreach` statement because the query itself must use `foreach` in order to return a result. Note also that these types of queries return a single value, not an `IEnumerable` collection. The following query returns a count of the even numbers in the source array:

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the [ToList](#) or [ToArray](#) methods.

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

You can also force execution by putting the `foreach` loop immediately after the query expression. However, by calling `ToList` or `ToArray` you also cache all the data in a single collection object.

See also

- [Getting Started with LINQ in C#](#)
- [Walkthrough: Writing Queries in C#](#)
- [Language Integrated Query \(LINQ\)](#)
- [foreach, in](#)
- [Query Keywords \(LINQ\)](#)

LINQ and Generic Types (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

LINQ queries are based on generic types, which were introduced in version 2.0 of .NET Framework. You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:

1. When you create an instance of a generic collection class such as `List<T>`, you replace the "T" with the type of objects that the list will hold. For example, a list of strings is expressed as `List<string>`, and a list of `Customer` objects is expressed as `List<Customer>`. A generic list is strongly typed and provides many benefits over collections that store their elements as `Object`. If you try to add a `Customer` to a `List<string>`, you will get an error at compile time. It is easy to use generic collections because you do not have to perform run-time type-casting.
2. `IEnumerable<T>` is the interface that enables generic collection classes to be enumerated by using the `foreach` statement. Generic collection classes support `IEnumerable<T>` just as non-generic collection classes such as `ArrayList` support `IEnumerable`.

For more information about generics, see [Generics](#).

`IEnumerable<T>` variables in LINQ Queries

LINQ query variables are typed as `IEnumerable<T>` or a derived type such as `IQueryable<T>`. When you see a query variable that is typed as `IEnumerable<Customer>`, it just means that the query, when it is executed, will produce a sequence of zero or more `Customer` objects.

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

For more information, see [Type Relationships in LINQ Query Operations](#).

Letting the Compiler Handle Generic Type Declarations

If you prefer, you can avoid generic syntax by using the `var` keyword. The `var` keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in the `from` clause. The following example produces the same compiled code as the previous example:

```
var customerQuery2 =  
    from cust in customers  
    where cust.City == "London"  
    select cust;  
  
foreach(var customer in customerQuery2)  
{  
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);  
}
```

The `var` keyword is useful when the type of the variable is obvious or when it is not that important to explicitly specify nested generic types such as those that are produced by group queries. In general, we recommend that if you use `var`, realize that it can make your code more difficult for others to read. For more information, see [Implicitly Typed Local Variables](#).

See also

- [Generics](#)

Basic LINQ Query Operations (C#)

11/2/2020 • 5 minutes to read • [Edit Online](#)

This topic gives a brief introduction to LINQ query expressions and some of the typical kinds of operations that you perform in a query. More detailed information is in the following topics:

[LINQ Query Expressions](#)

[Standard Query Operators Overview \(C#\)](#)

[Walkthrough: Writing Queries in C#](#)

NOTE

If you already are familiar with a query language such as SQL or XQuery, you can skip most of this topic. Read about the "from clause" in the next section to learn about the order of clauses in LINQ query expressions.

Obtaining a Data Source

In a LINQ query, the first step is to specify the data source. In C# as in most programming languages a variable must be declared before it can be used. In a LINQ query, the `from` clause comes first in order to introduce the data source (`customers`) and the *range variable* (`cust`).

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable will serve as a reference to each successive element in `customers`. Because the compiler can infer the type of `cust`, you do not have to specify it explicitly. Additional range variables can be introduced by a `let` clause. For more information, see [let clause](#).

NOTE

For non-generic data sources such as [ArrayList](#), the range variable must be explicitly typed. For more information, see [How to query an ArrayList with LINQ \(C#\)](#) and `from` clause.

Filtering

Probably the most common query operation is to apply a filter in the form of a Boolean expression. The filter causes the query to return only those elements for which the expression is true. The result is produced by using the `where` clause. The filter in effect specifies which elements to exclude from the source sequence. In the following example, only those `customers` who have an address in London are returned.

```
var queryLondonCustomers = from cust in customers
                            where cust.City == "London"
                            select cust;
```

You can use the familiar C# logical `AND` and `OR` operators to apply as many filter expressions as necessary in

the `where` clause. For example, to return only customers from "London" AND whose name is "Devon" you would write the following code:

```
where cust.City == "London" && cust.Name == "Devon"
```

To return customers from London or Paris, you would write the following code:

```
where cust.City == "London" || cust.City == "Paris"
```

For more information, see [where clause](#).

Ordering

Often it is convenient to sort the returned data. The `orderby` clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example, the following query can be extended to sort the results based on the `Name` property. Because `Name` is a string, the default comparer performs an alphabetical sort from A to Z.

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

To order the results in reverse order, from Z to A, use the `orderby...descending` clause.

For more information, see [orderby clause](#).

Grouping

The `group` clause enables you to group your results based on a key that you specify. For example you could specify that the results should be grouped by the `City` so that all customers from London or Paris are in individual groups. In this case, `cust.City` is the key.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

When you end a query with a `group` clause, your results take the form of a list of lists. Each element in the list is an object that has a `Key` member and a list of elements that are grouped under that key. When you iterate over a query that produces a sequence of groups, you must use a nested `foreach` loop. The outer loop iterates over each group, and the inner loop iterates over each group's members.

If you must refer to the results of a group operation, you can use the `into` keyword to create an identifier that

can be queried further. The following query returns only those groups that contain more than two customers:

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

For more information, see [group clause](#).

Joining

Join operations create associations between sequences that are not explicitly modeled in the data sources. For example you can perform a join to find all the customers and distributors who have the same location. In LINQ the `join` clause always works against object collections instead of database tables directly.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

In LINQ you do not have to use `join` as often as you do in SQL, because foreign keys in LINQ are represented in the object model as properties that hold a collection of items. For example, a `Customer` object contains a collection of `Order` objects. Rather than performing a join, you access the orders by using dot notation:

```
from order in Customer.Orders...
```

For more information, see [join clause](#).

Selecting (Projections)

The `select` clause produces the results of the query and specifies the "shape" or type of each returned element. For example, you can specify whether your results will consist of complete `Customer` objects, just one member, a subset of members, or some completely different result type based on a computation or new object creation. When the `select` clause produces something other than a copy of the source element, the operation is called a *projection*. The use of projections to transform data is a powerful capability of LINQ query expressions. For more information, see [Data Transformations with LINQ \(C#\)](#) and [select clause](#).

See also

- [LINQ Query Expressions](#)
- [Walkthrough: Writing Queries in C#](#)
- [Query Keywords \(LINQ\)](#)
- [Anonymous Types](#)

Data Transformations with LINQ (C#)

11/2/2020 • 6 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is not only about retrieving data. It is also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping. But perhaps the most powerful feature of LINQ queries is the ability to create new types. This is accomplished in the `select` clause. For example, you can perform the following tasks:

- Merge multiple input sequences into a single output sequence that has a new type.
- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

These are just several examples. Of course, these transformations can be combined in various ways in the same query. Furthermore, the output sequence of one query can be used as the input sequence for a new query.

Joining Multiple Inputs into One Output Sequence

You can use a LINQ query to create an output sequence that contains elements from more than one input sequence. The following example shows how to combine two in-memory data structures, but the same principles can be applied to combine data from XML or SQL or DataSet sources. Assume the following two class types:

```
class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

The following example shows the query:

```

class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };
    }

    // Create the second data source.
    List<Teacher> teachers = new List<Teacher>()
    {
        new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
        new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
        new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
    };

    // Create the query.
    var peopleInSeattle = (from student in students
                           where student.City == "Seattle"
                           select student.Last)
                           .Concat(from teacher in teachers
                                   where teacher.City == "Seattle"
                                   select teacher.Last);

    Console.WriteLine("The following students and teachers live in Seattle:");
    // Execute the query.
    foreach (var person in peopleInSeattle)
    {
        Console.WriteLine(person);
    }

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
   The following students and teachers live in Seattle:
   Omelchenko
   Beebe
*/

```

For more information, see [join clause](#) and [select clause](#).

Selecting a Subset of each Source Element

There are two primary ways to select a subset of each element in the source sequence:

1. To select just one member of the source element, use the dot operation. In the following example, assume that a `Customer` object contains several public properties including a string named `city`. When executed, this query will produce an output sequence of strings.

```
var query = from cust in Customers
            select cust.City;
```

2. To create elements that contain more than one property from the source element, you can use an object initializer with either a named object or an anonymous type. The following example shows the use of an anonymous type to encapsulate two properties from each `Customer` element:

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

For more information, see [Object and Collection Initializers](#) and [Anonymous Types](#).

Transforming in-Memory Objects into XML

LINQ queries make it easy to transform data between in-memory data structures, SQL databases, ADO.NET Datasets and XML streams or documents. The following example transforms objects in an in-memory data structure into XML elements.

```
class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(", ", student.Scores)
            select new XElement("student",
                new XElement("First", student.First),
                new XElement("Last", student.Last),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

The code produces the following XML output:

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

For more information, see [Creating XML Trees in C# \(LINQ to XML\)](#).

Performing Operations on Source Elements

An output sequence might not contain any elements or element properties from the source sequence. The output might instead be a sequence of values that is computed by using the source elements as input arguments.

The following query will take a sequence of numbers that represent radii of circles, calculate the area for each radius, and return an output sequence containing strings formatted with the calculated area.

Each string for the output sequence will be formatted using [string interpolation](#). An interpolated string will have a \$ in front of the string's opening quotation mark, and operations can be performed within curly braces placed inside the interpolated string. Once those operations are performed, the results will be concatenated.

NOTE

Calling methods in query expressions is not supported if the query will be translated into some other domain. For example, you cannot call an ordinary C# method in LINQ to SQL because SQL Server has no context for it. However, you can map stored procedures to methods and call those. For more information, see [Stored Procedures](#).

```

class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        I Enumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Area for a circle with a radius of '1' = 3.14
   Area for a circle with a radius of '2' = 12.57
   Area for a circle with a radius of '3' = 28.27
*/

```

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [LINQ to SQL](#)
- [LINQ to DataSet](#)
- [LINQ to XML \(C#\)](#)
- [LINQ Query Expressions](#)
- [select clause](#)

Type Relationships in LINQ Query Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

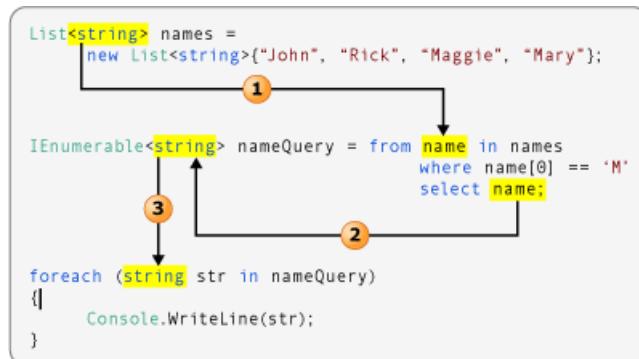
To write queries effectively, you should understand how types of the variables in a complete query operation all relate to each other. If you understand these relationships you will more easily comprehend the LINQ samples and code examples in the documentation. Furthermore, you will understand what occurs behind the scenes when variables are implicitly typed by using `var`.

LINQ query operations are strongly typed in the data source, in the query itself, and in the query execution. The type of the variables in the query must be compatible with the type of the elements in the data source and with the type of the iteration variable in the `foreach` statement. This strong typing guarantees that type errors are caught at compile time when they can be corrected before users encounter them.

In order to demonstrate these type relationships, most of the examples that follow use explicit typing for all variables. The last example shows how the same principles apply even when you use implicit typing by using `var`.

Queries that do not Transform the Source Data

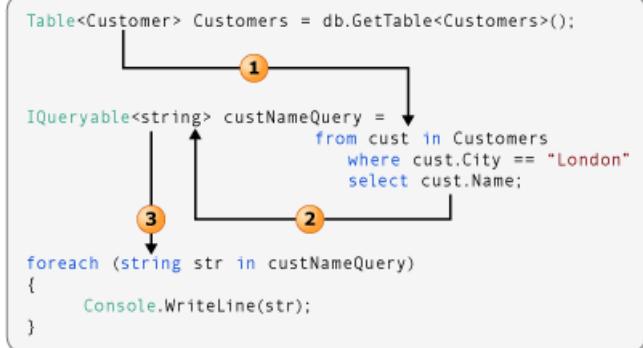
The following illustration shows a LINQ to Objects query operation that performs no transformations on the data. The source contains a sequence of strings and the query output is also a sequence of strings.



1. The type argument of the data source determines the type of the range variable.
2. The type of the object that is selected determines the type of the query variable. Here `name` is a string. Therefore, the query variable is an `IEnumerable<string>`.
3. The query variable is iterated over in the `foreach` statement. Because the query variable is a sequence of strings, the iteration variable is also a string.

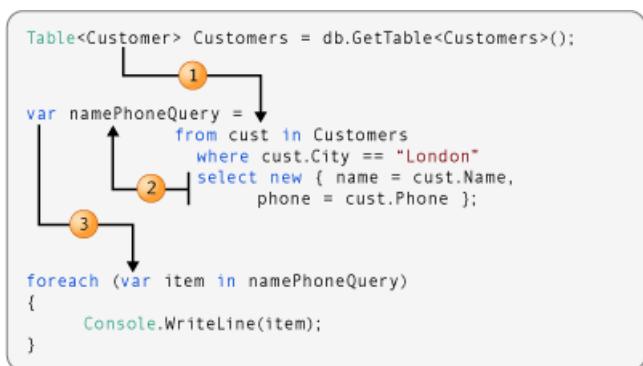
Queries that Transform the Source Data

The following illustration shows a LINQ to SQL query operation that performs a simple transformation on the data. The query takes a sequence of `Customer` objects as input, and selects only the `Name` property in the result. Because `Name` is a string, the query produces a sequence of strings as output.



1. The type argument of the data source determines the type of the range variable.
2. The `select` statement returns the `Name` property instead of the complete `Customer` object. Because `Name` is a string, the type argument of `custNameQuery` is `string`, not `Customer`.
3. Because `custNameQuery` is a sequence of strings, the `foreach` loop's iteration variable must also be a `string`.

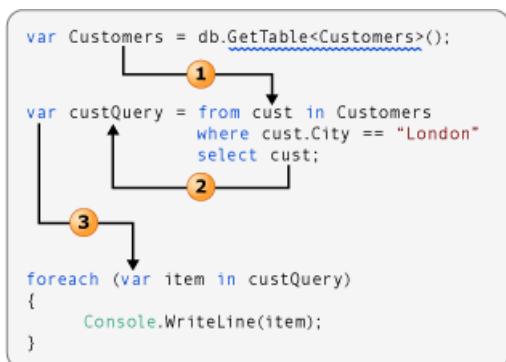
The following illustration shows a slightly more complex transformation. The `select` statement returns an anonymous type that captures just two members of the original `Customer` object.



1. The type argument of the data source is always the type of the range variable in the query.
2. Because the `select` statement produces an anonymous type, the query variable must be implicitly typed by using `var`.
3. Because the type of the query variable is implicit, the iteration variable in the `foreach` loop must also be implicit.

Letting the compiler infer type information

Although you should understand the type relationships in a query operation, you have the option to let the compiler do all the work for you. The keyword `var` can be used for any local variable in a query operation. The following illustration is similar to example number 2 that was discussed earlier. However, the compiler supplies the strong type for each variable in the query operation.



For more information about `var`, see [Implicitly Typed Local Variables](#).

Query Syntax and Method Syntax in LINQ (C#)

11/2/2020 • 4 minutes to read • [Edit Online](#)

Most queries in the introductory Language Integrated Query (LINQ) documentation are written by using the LINQ declarative query syntax. However, the query syntax must be translated into method calls for the .NET common language runtime (CLR) when the code is compiled. These method calls invoke the standard query operators, which have names such as `Where`, `Select`, `GroupBy`, `Join`, `Max`, and `Average`. You can call them directly by using method syntax instead of query syntax.

Query syntax and method syntax are semantically identical, but many people find query syntax simpler and easier to read. Some queries must be expressed as method calls. For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence. The reference documentation for the standard query operators in the `System.Linq` namespace generally uses method syntax. Therefore, even when getting started writing LINQ queries, it is useful to be familiar with how to use method syntax in queries and in query expressions themselves.

Standard Query Operator Extension Methods

The following example shows a simple *query expression* and the semantically equivalent query written as a *method-based query*.

```

class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

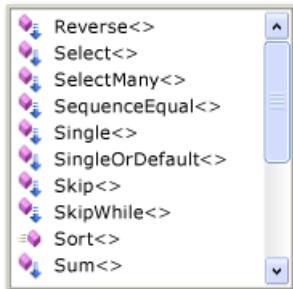
        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
Output:
6 8 10 12
6 8 10 12
*/

```

The output from the two examples is identical. You can see that the type of the query variable is the same in both forms: `IEnumerable<T>`.

To understand the method-based query, let's examine it more closely. On the right side of the expression, notice that the `where` clause is now expressed as an instance method on the `numbers` object, which as you will recall has a type of `IEnumerable<int>`. If you are familiar with the generic `IEnumerable<T>` interface, you know that it does not have a `Where` method. However, if you invoke the IntelliSense completion list in the Visual Studio IDE, you will see not only a `Where` method, but many other methods such as `Select`, `SelectMany`, `Join`, and `OrderBy`. These are all the standard query operators.

```
List<string> list = new List<string>();
list.|
```



Although it looks as if `IEnumerable<T>` has been redefined to include these additional methods, in fact this is not the case. The standard query operators are implemented as a new kind of method called *extension methods*. Extension methods "extend" an existing type; they can be called as if they were instance methods on the type.

The standard query operators extend `IEnumerable<T>` and that is why you can write `numbers.Where(...)`.

To get started using LINQ all that you really have to know about extension methods is how to bring them into scope in your application by using the correct `using` directives. From your application's point of view, an extension method and a regular instance method are the same.

For more information about extension methods, see [Extension Methods](#). For more information about standard query operators, see [Standard Query Operators Overview \(C#\)](#). Some LINQ providers, such as LINQ to SQL and LINQ to XML, implement their own standard query operators and additional extension methods for other types besides `IEnumerable<T>`.

Lambda Expressions

In the previous example, notice that the conditional expression (`num % 2 == 0`) is passed as an in-line argument to the `Where` method: `Where(num => num % 2 == 0)`. This inline expression is called a lambda expression. It is a convenient way to write code that would otherwise have to be written in more cumbersome form as an anonymous method or a generic delegate or an expression tree. In C# `=>` is the lambda operator, which is read as "goes to". The `num` on the left of the operator is the input variable which corresponds to `num` in the query expression. The compiler can infer the type of `num` because it knows that `numbers` is a generic `IEnumerable<T>` type. The body of the lambda is just the same as the expression in query syntax or in any other C# expression or statement; it can include method calls and other complex logic. The "return value" is just the expression result.

To get started using LINQ you do not have to use lambdas extensively. However, certain queries can only be expressed in method syntax and some of those require lambda expressions. After you become more familiar with lambdas, you will find that they are a powerful and flexible tool in your LINQ toolbox. For more information, see [Lambda Expressions](#).

Composability of Queries

In the previous code example, note that the `OrderBy` method is invoked by using the dot operator on the call to `Where`. `Where` produces a filtered sequence, and then `orderby` operates on that sequence by sorting it. Because queries return an `IEnumerable`, you compose them in method syntax by chaining the method calls together. This is what the compiler does behind the scenes when you write queries by using query syntax. And because a query variable does not store the results of the query, you can modify it or use it as the basis for a new query at any time, even after it has been executed.

C# Features That Support LINQ

11/2/2020 • 3 minutes to read • [Edit Online](#)

The following section introduces new language constructs introduced in C# 3.0. Although these new features are all used to a degree with LINQ queries, they are not limited to LINQ and can be used in any context where you find them useful.

Query Expressions

Query expressions use a declarative syntax similar to SQL or XQuery to query over `IEnumerable` collections. At compile time query syntax is converted to method calls to a LINQ provider's implementation of the standard query operator extension methods. Applications control the standard query operators that are in scope by specifying the appropriate namespace with a `using` directive. The following query expression takes an array of strings, groups them according to the first character in the string, and orders the groups.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

For more information, see [LINQ Query Expressions](#).

Implicitly Typed Variables (`var`)

Instead of explicitly specifying a type when you declare and initialize a variable, you can use the `var` modifier to instruct the compiler to infer and assign the type, as shown here:

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Variables declared as `var` are just as strongly typed as variables whose type you specify explicitly. The use of `var` makes it possible to create anonymous types, but it can be used only for local variables. Arrays can also be declared with implicit typing.

For more information, see [Implicitly Typed Local Variables](#).

Object and Collection Initializers

Object and collection initializers make it possible to initialize objects without explicitly calling a constructor for the object. Initializers are typically used in query expressions when they project the source data into a new data type. Assuming a class named `Customer` with public `Name` and `Phone` properties, the object initializer can be used as in the following code:

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuing with our `Customer` class, assume that there is a data source called `IncomingOrders`, and that for each order with a large `OrderSize`, we would like to create a new `Customer` based off of that order. A LINQ query can

be executed on this data source and use object initialization to fill a collection:

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

The data source may have more properties lying under the hood than the `Customer` class such as `OrderSize`, but with object initialization, the data returned from the query is molded into the desired data type; we choose the data that is relevant to our class. As a result, we now have an `IEnumerable` filled with the new `Customer`s we wanted. The above can also be written in LINQ's method syntax:

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

For more information, see:

- [Object and Collection Initializers](#)
- [Query Expression Syntax for Standard Query Operators](#)

Anonymous Types

An anonymous type is constructed by the compiler and the type name is only available to the compiler. Anonymous types provide a convenient way to group a set of properties temporarily in a query result without having to define a separate named type. Anonymous types are initialized with a new expression and an object initializer, as shown here:

```
select new {name = cust.Name, phone = cust.Phone};
```

For more information, see [Anonymous Types](#).

Extension Methods

An extension method is a static method that can be associated with a type, so that it can be called as if it were an instance method on the type. This feature enables you to, in effect, "add" new methods to existing types without actually modifying them. The standard query operators are a set of extension methods that provide LINQ query functionality for any type that implements `IEnumerable<T>`.

For more information, see [Extension Methods](#).

Lambda Expressions

A lambda expression is an inline function that uses the `=>` operator to separate input parameters from the function body and can be converted at compile time to a delegate or an expression tree. In LINQ programming, you will encounter lambda expressions when you make direct method calls to the standard query operators.

For more information, see:

- [Anonymous Functions](#)
- [Lambda Expressions](#)
- [Expression Trees \(C#\)](#)

See also

- Language-Integrated Query (LINQ) (C#)

Walkthrough: Writing Queries in C# (LINQ)

11/2/2020 • 10 minutes to read • [Edit Online](#)

This walkthrough demonstrates the C# language features that are used to write LINQ query expressions.

Create a C# Project

NOTE

The following instructions are for Visual Studio. If you are using a different development environment, create a console project with a reference to System.Core.dll and a `using` directive for the `System.Linq` namespace.

To create a project in Visual Studio

1. Start Visual Studio.
2. On the menu bar, choose **File**, **New**, **Project**.
The **New Project** dialog box opens.
3. Expand **Installed**, expand **Templates**, expand **Visual C#**, and then choose **Console Application**.
4. In the **Name** text box, enter a different name or accept the default name, and then choose the **OK** button.
The new project appears in **Solution Explorer**.
5. Notice that your project has a reference to System.Core.dll and a `using` directive for the `System.Linq` namespace.

Create an in-Memory Data Source

The data source for the queries is a simple list of `Student` objects. Each `Student` record has a first name, last name, and an array of integers that represents their test scores in the class. Copy this code into your project. Note the following characteristics:

- The `Student` class consists of auto-implemented properties.
- Each student in the list is initialized with an object initializer.
- The list itself is initialized with a collection initializer.

This whole data structure will be initialized and instantiated without explicit calls to any constructor or explicit member access. For more information about these new features, see [Auto-Implemented Properties](#) and [Object and Collection Initializers](#).

To add the data source

- Add the `Student` class and the initialized list of students to the `Program` class in your project.

```

public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};

```

To add a new Student to the Students list

1. Add a new `student` to the `Students` list and use a name and test scores of your choice. Try typing all the new student information in order to better learn the syntax for the object initializer.

Create the Query

To create a simple query

- In the application's `Main` method, create a simple query that, when it is executed, will produce a list of all students whose score on the first test was greater than 90. Note that because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the `var` keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see [Implicitly Typed Local Variables](#).)

Note also that the query's range variable, `student`, serves as a reference to each `student` in the source, providing member access for each object.

```

// Create the query.
// The first line could also be written as "var studentQuery ="
IQueryable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;

```

Execute the Query

To execute the query

1. Now write the `foreach` loop that will cause the query to execute. Note the following about the code:

- Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop.
- The type of this variable is `student`, and the type of the query variable is compatible,

```
IEnumerable<Student> .
```

- After you have added this code, build and run the application to see the results in the **Console** window.

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine("{0}, {1}", student.Last, student.First);  
}  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry  
// Zabokritski, Eugene  
// Tucker, Michael
```

To add another filter condition

- You can combine multiple Boolean conditions in the `where` clause in order to further refine a query. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

For more information, see [where clause](#).

Modify the Query

To order the results

- It will be easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the last name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

```
orderby student.Last ascending
```

- Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

```
orderby student.Scores[0] descending
```

- Change the `WriteLine` format string so that you can see the scores:

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

For more information, see [orderby clause](#).

To group the results

- Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their last name as the key.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

2. Note that the type of the query has now changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `Student` objects. Because the type of the query has changed, the following code changes the `foreach` execution loop also:

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}",
                          student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

3. Run the application and view the results in the **Console** window.

For more information, see [group clause](#).

To make the variables implicitly typed

1. Explicitly coding `IEnumerables` or `IGroupings` can quickly become tedious. You can write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword does not change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. Note that in the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works just as before. Change the `s` iteration variable to `var` and run the query again. You see that you get exactly the same results.

```

var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine(" {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene

```

For more information about `var`, see [Implicitly Typed Local Variables](#).

To order the groups by their key value

- When you run the previous query, you notice that the groups are not in alphabetical order. To change this, you must provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:

```

var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine(" {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
// Adams, Terry
//F
// Fakhouri, Fadi
// Feng, Hanying
//G
// Garcia, Cesar
// Garcia, Debra
// Garcia, Hugo
//M
// Mortensen, Sven
//O
// Omelchenko, Svetlana
// O'Donnell, Claire
//T
// Tucker, Lance
// Tucker, Michael
//Z
// Zabokritski, Eugene

```

When you run this query, you will see the groups are now sorted in alphabetical order.

To introduce an identifier by using let

1. You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example, or it can enhance performance by storing the results of an expression so that it does not have to be calculated multiple times.

```

// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael

```

For more information, see [let clause](#).

To use method syntax in a query expression

- As described in [Query Syntax and Method Syntax in LINQ](#), some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

```

var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667

```

To transform or project in the select clause

- It is very common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. Note that the query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.

```

IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo

```

2. Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `Students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

```

var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368

```

Next Steps

After you are familiar with the basic aspects of working with queries in C#, you are ready to read the documentation and samples for the specific type of LINQ provider you are interested in:

[LINQ to SQL](#)

[LINQ to DataSet](#)

[LINQ to XML \(C#\)](#)

[LINQ to Objects \(C#\)](#)

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [LINQ Query Expressions](#)

Standard Query Operators Overview (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The *standard query operators* are the methods that form the LINQ pattern. Most of these methods operate on sequences, where a sequence is an object whose type implements the `IEnumerable<T>` interface or the `IQueryable<T>` interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting and more.

There are two sets of LINQ standard query operators: one that operates on objects of type `IEnumerable<T>`, another that operates on objects of type `IQueryable<T>`. The methods that make up each set are static members of the `Enumerable` and `Queryable` classes, respectively. They are defined as *extension methods* of the type that they operate on. Extension methods can be called by using either static method syntax or instance method syntax.

In addition, several standard query operator methods operate on types other than those based on `IEnumerable<T>` or `IQueryable<T>`. The `Enumerable` type defines two such methods that both operate on objects of type `IEnumerable`. These methods, `Cast<TResult>(IEnumerable)` and `OfType<TResult>(IEnumerable)`, let you enable a non-parameterized, or non-generic, collection to be queried in the LINQ pattern. They do this by creating a strongly typed collection of objects. The `Queryable` class defines two similar methods, `Cast<TResult>(IQueryable)` and `OfType<TResult>(IQueryable)`, that operate on objects of type `Queryable`.

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that return a singleton value (for example, `Average` and `Sum`) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object.

For methods that operate on in-memory collections, that is, those methods that extend `IEnumerable<T>`, the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

In contrast, methods that extend `IQueryable<T>` don't implement any querying behavior. They build an expression tree that represents the query to be performed. The query processing is handled by the source `IQueryable<T>` object.

Calls to query methods can be chained together in one query, which enables queries to become arbitrarily complex.

The following code example demonstrates how the standard query operators can be used to obtain information about a sequence.

```

string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS

```

Query Expression Syntax

Some of the more frequently used standard query operators have dedicated C# and Visual Basic language keyword syntax that enables them to be called as part of a *query expression*. For more information about standard query operators that have dedicated keywords and their corresponding syntaxes, see [Query Expression Syntax for Standard Query Operators \(C#\)](#).

Extending the Standard Query Operators

You can augment the set of standard query operators by creating domain-specific methods that are appropriate for your target domain or technology. You can also replace the standard query operators with your own implementations that provide additional services such as remote evaluation, query translation, and optimization. See [AsEnumerable](#) for an example.

Related Sections

The following links take you to articles that provide additional information about the various standard query operators based on functionality.

[Sorting Data \(C#\)](#)

[Set Operations \(C#\)](#)

[Filtering Data \(C#\)](#)

[Quantifier Operations \(C#\)](#)

[Projection Operations \(C#\)](#)

[Partitioning Data \(C#\)](#)

[Join Operations \(C#\)](#)

[Grouping Data \(C#\)](#)

[Generation Operations \(C#\)](#)

[Equality Operations \(C#\)](#)

[Element Operations \(C#\)](#)

[Converting Data Types \(C#\)](#)

[Concatenation Operations \(C#\)](#)

[Aggregation Operations \(C#\)](#)

See also

- [Enumerable](#)
- [Queryable](#)
- [Introduction to LINQ Queries \(C#\)](#)
- [Query Expression Syntax for Standard Query Operators \(C#\)](#)
- [Classification of Standard Query Operators by Manner of Execution \(C#\)](#)
- [Extension Methods](#)

Query Expression Syntax for Standard Query Operators (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Some of the more frequently used standard query operators have dedicated C# language keyword syntax that enables them to be called as part of a *query expression*. A query expression is a different, more readable form of expressing a query than its *method-based* equivalent. Query expression clauses are translated into calls to the query methods at compile time.

Query Expression Syntax Table

The following table lists the standard query operators that have equivalent query expression clauses.

METHOD	C# QUERY EXPRESSION SYNTAX
Cast	Use an explicitly typed range variable, for example: <code>from int i in numbers</code> (For more information, see from clause .)
GroupBy	<code>group ... by</code> -or- <code>group ... by ... into ...</code> (For more information, see group clause .)
GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	<code>join ... in ... on ... equals ... into ...</code> (For more information, see join clause .)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	<code>join ... in ... on ... equals ...</code> (For more information, see join clause .)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby</code> (For more information, see orderby clause .)
OrderByDescending<TSource,TKey> (IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... descending</code> (For more information, see orderby clause .)
Select	<code>select</code> (For more information, see select clause .)

METHOD	C# QUERY EXPRESSION SYNTAX
SelectMany	<p>Multiple <code>from</code> clauses.</p> <p>(For more information, see from clause.)</p>
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... , ...</code> <p>(For more information, see orderby clause.)</p>
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... , ... descending</code> <p>(For more information, see orderby clause.)</p>
Where	<code>where</code> <p>(For more information, see where clause.)</p>

See also

- [Enumerable](#)
- [Queryable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Classification of Standard Query Operators by Manner of Execution \(C#\)](#)

Classification of Standard Query Operators by Manner of Execution (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The LINQ to Objects implementations of the standard query operator methods execute in one of two main ways: immediate or deferred. The query operators that use deferred execution can be additionally divided into two categories: streaming and non-streaming. If you know how the different query operators execute, it may help you understand the results that you get from a given query. This is especially true if the data source is changing or if you are building a query on top of another query. This topic classifies the standard query operators according to their manner of execution.

Manners of Execution

Immediate

Immediate execution means that the data source is read and the operation is performed at the point in the code where the query is declared. All the standard query operators that return a single, non-enumerable result execute immediately.

Deferred

Deferred execution means that the operation is not performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a `foreach` statement. This means that the results of executing the query depend on the contents of the data source when the query is executed rather than when the query is defined. If the query variable is enumerated multiple times, the results might differ every time. Almost all the standard query operators whose return type is `IEnumerable<T>` or `IOrderedEnumerable<TElement>` execute in a deferred manner.

Query operators that use deferred execution can be additionally classified as streaming or non-streaming.

Streaming

Streaming operators do not have to read all the source data before they yield elements. At the time of execution, a streaming operator performs its operation on each source element as it is read and yields the element if appropriate. A streaming operator continues to read source elements until a result element can be produced. This means that more than one source element might be read to produce one result element.

Non-Streaming

Non-streaming operators must read all the source data before they can yield a result element. Operations such as sorting or grouping fall into this category. At the time of execution, non-streaming query operators read all the source data, put it into a data structure, perform the operation, and yield the resulting elements.

Classification Table

The following table classifies each standard query operator method according to its method of execution.

NOTE

If an operator is marked in two columns, two input sequences are involved in the operation, and each sequence is evaluated differently. In these cases, it is always the first sequence in the parameter list that is evaluated in a deferred, streaming manner.

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Single numeric value	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		
LastOrDefault	TSource	X		
LongCount	Int64	X		

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
Max	Single numeric value, TSource, or TResult	X		
Min	Single numeric value, TSource, or TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Single numeric value	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	TSource array	X		
ToDictionary	Dictionary< TKey, TValue >	X		

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
ToList	IList<T>	X		
ToLookup	ILookup< TKey, TElement >	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

See also

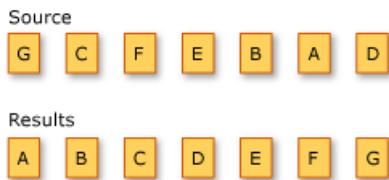
- [Enumerable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Query Expression Syntax for Standard Query Operators \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

Sorting Data (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A sorting operation orders the elements of a sequence based on one or more attributes. The first sort criterion performs a primary sort on the elements. By specifying a second sort criterion, you can sort the elements within each primary sort group.

The following illustration shows the results of an alphabetical sort operation on a sequence of characters:



The standard query operator methods that sort data are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
OrderBy	Sorts values in ascending order.	<code>orderby</code>	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Sorts values in descending order.	<code>orderby ... descending</code>	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Performs a secondary sort in ascending order.	<code>orderby ..., ...</code>	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Performs a secondary sort in descending order.	<code>orderby ..., ... descending</code>	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Reverses the order of the elements in a collection.	Not applicable.	Enumerable.Reverse Queryable.Reverse

Query Expression Syntax Examples

Primary Sort Examples

Primary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to sort the strings in an array by string length, in ascending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                             orderby word.Length
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
fox
quick
brown
jumps
*/

```

Primary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to sort the strings by their first letter, in descending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                             orderby word.Substring(0, 1) descending
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
quick
jumps
fox
brown
*/

```

Secondary Sort Examples

Secondary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to perform a primary and secondary sort of the strings in an array. The strings are sorted primarily by length and secondarily by the first letter of the string, both in ascending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1)
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/

```

Secondary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to perform a primary sort, in ascending order, and a secondary sort, in descending order. The strings are sorted primarily by length and secondarily by the first letter of the string.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/

```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [orderby clause](#)
- [Order the results of a join clause](#)
- [How to sort or filter text data by any word or field \(LINQ\) \(C#\)](#)

Set Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set operations in LINQ refer to query operations that produce a result set that is based on the presence or absence of equivalent elements within the same or separate collections (or sets).

The standard query operator methods that perform set operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Distinct	Removes duplicate values from a collection.	Not applicable.	Enumerable.Distinct Queryable.Distinct
Except	Returns the set difference, which means the elements of one collection that do not appear in a second collection.	Not applicable.	Enumerable.Except Queryable.Except
Intersect	Returns the set intersection, which means elements that appear in each of two collections.	Not applicable.	Enumerable.Intersect Queryable.Intersect
Union	Returns the set union, which means unique elements that appear in either of two collections.	Not applicable.	Enumerable.Union Queryable.Union

Comparison of Set Operations

Distinct

The following example depicts the behavior of the [Enumerable.Distinct](#) method on a sequence of characters. The returned sequence contains the unique elements from the input sequence.



```

string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                             select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */

```

Except

The following example depicts the behavior of [Enumerable.Except](#). The returned sequence contains only the elements from the first input sequence that are not in the second input sequence.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                             select planet;

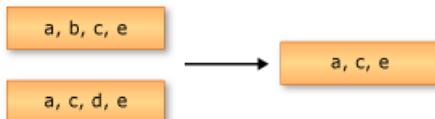
foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */

```

Intersect

The following example depicts the behavior of [Enumerable.Intersect](#). The returned sequence contains the elements that are common to both of the input sequences.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IQueryable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;

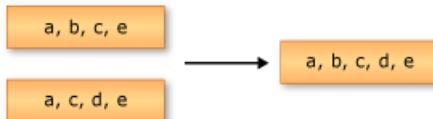
foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */

```

Union

The following example depicts a union operation on two sequences of characters. The returned sequence contains the unique elements from both input sequences.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IQueryable<string> query = from planet in planets1.Union(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */

```

See also

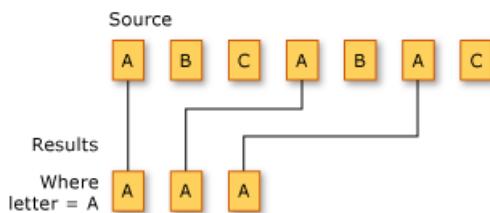
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to combine and compare string collections \(LINQ\) \(C#\)](#)
- [How to find the set difference between two lists \(LINQ\) \(C#\)](#)

Filtering Data (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Filtering refers to the operation of restricting the result set to contain only those elements that satisfy a specified condition. It is also known as selection.

The following illustration shows the results of filtering a sequence of characters. The predicate for the filtering operation specifies that the character must be 'A'.



The standard query operator methods that perform selection are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
OfType	Selects values, depending on their ability to be cast to a specified type.	Not applicable.	Enumerable.OfType Queryable.OfType
Where	Selects values that are based on a predicate function.	where	Enumerable.Where Queryable.Where

Query Expression Syntax Example

The following example uses the `where` clause to filter from an array those strings that have a specific length.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:
   the
   fox
*/
```

See also

- [System.Linq](#)

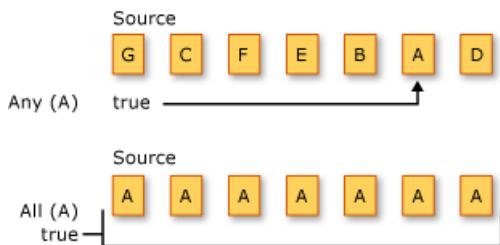
- [Standard Query Operators Overview \(C#\)](#)
- [where clause](#)
- [Dynamically specify predicate filters at runtime](#)
- [How to query an assembly's metadata with Reflection \(LINQ\) \(C#\)](#)
- [How to query for files with a specified attribute or name \(C#\)](#)
- [How to sort or filter text data by any word or field \(LINQ\) \(C#\)](#)

Quantifier Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Quantifier operations return a [Boolean](#) value that indicates whether some or all of the elements in a sequence satisfy a condition.

The following illustration depicts two different quantifier operations on two different source sequences. The first operation asks if one or more of the elements are the character 'A', and the result is `true`. The second operation asks if all the elements are the character 'A', and the result is `false`.



The standard query operator methods that perform quantifier operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
All	Determines whether all the elements in a sequence satisfy a condition.	Not applicable.	Enumerable.All Queryable.All
Any	Determines whether any elements in a sequence satisfy a condition.	Not applicable.	Enumerable.Any Queryable.Any
Contains	Determines whether a sequence contains a specified element.	Not applicable.	Enumerable.Contains Queryable.Contains

Query Expression Syntax Examples

All

The following example uses the `All` to check that all strings are of a specific length.

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                where market.Items.All(item => item.Length == 5)
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}

```

Any

The following example uses the `Any` to check that any strings are start with 'o'.

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                where market.Items.Any(item => item.StartsWith("o"))
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}

```

Contains

The following example uses the `Contains` to check an array have a specific element.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                  where market.Items.Contains("kiwi")
                                  select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Dynamically specify predicate filters at runtime](#)
- [How to query for sentences that contain a specified set of words \(LINQ\) \(C#\)](#)

Projection Operations (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Projection refers to the operation of transforming an object into a new form that often consists only of those properties that will be subsequently used. By using projection, you can construct a new type that is built from each object. You can project a property and perform a mathematical function on it. You can also project the original object without changing it.

The standard query operator methods that perform projection are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Select	Projects values that are based on a transform function.	<code>select</code>	Enumerable.Select Queryable.Select
SelectMany	Projects sequences of values that are based on a transform function and then flattens them into one sequence.	Use multiple <code>from</code> clauses	Enumerable.SelectMany Queryable.SelectMany

Query Expression Syntax Examples

Select

The following example uses the `select` clause to project the first letter from each string in a list of strings.

```
List<string> words = new List<string>() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

The following example uses multiple `from` clauses to project each word from each string in a list of strings.

```

List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

an
apple
a
day
the
quick
brown
fox
*/

```

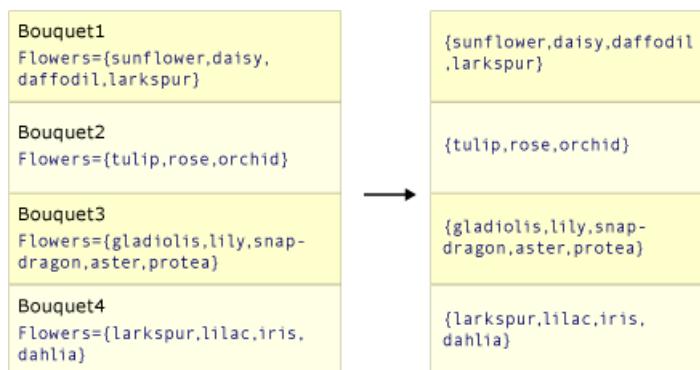
Select versus SelectMany

The work of both `Select()` and `SelectMany()` is to produce a result value (or values) from source values.

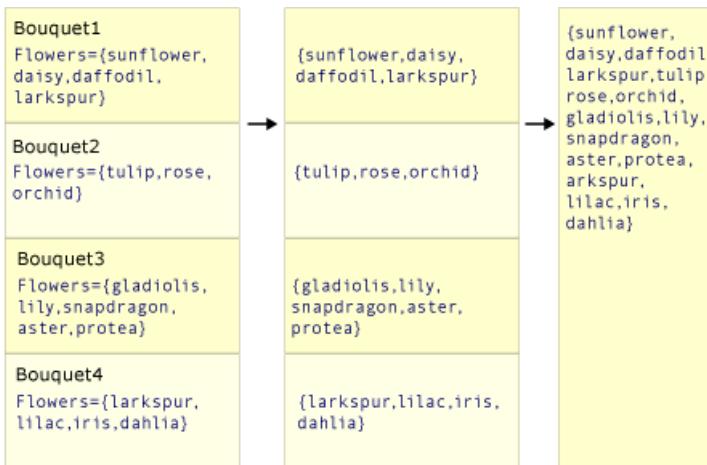
`Select()` produces one result value for every source value. The overall result is therefore a collection that has the same number of elements as the source collection. In contrast, `SelectMany()` produces a single overall result that contains concatenated sub-collections from each source value. The transform function that is passed as an argument to `SelectMany()` must return an enumerable sequence of values for each source value. These enumerable sequences are then concatenated by `SelectMany()` to create one large sequence.

The following two illustrations show the conceptual difference between the actions of these two methods. In each case, assume that the selector (transform) function selects the array of flowers from each source value.

This illustration depicts how `Select()` returns a collection that has the same number of elements as the source collection.



This illustration depicts how `SelectMany()` concatenates the intermediate sequence of arrays into one final result value that contains each value from each intermediate array.



Code Example

The following example compares the behavior of `Select()` and `SelectMany()`. The code creates a "bouquet" of flowers by taking the first two items from each list of flower names in the source collection. In this example, the "single value" that the transform function `Select<TSource,TResult>(IQueryable<TSource>, Func<TSource,TResult>)` uses is itself a collection of values. This requires the extra `foreach` loop in order to enumerate each string in each sub-sequence.

```

class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new List<Bouquet>()
    {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" } },
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid" } },
        new Bouquet { Flowers = new List<string> { "gladiolus", "lily", "snapdragon", "aster", "protea" } },
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" } }
    };

    // ***** Select *****
    I Enumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    // ***** SelectMany *****
    I Enumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (I Enumerable<String> collection in query1)
        foreach (string item in collection)
            Console.WriteLine(item);

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
        Console.WriteLine(item);

    /* This code produces the following output:

    Results by using Select():
    sunflower
    daisy
    daffodil
    larkspur
    tulip
    rose
    orchid
    gladiolus
    lily
    snapdragon
    */
}

```

```
Snipper agviii
aster
protea
larkspur
lilac
iris
dahlia

Results by using SelectMany():
sunflower
daisy
daffodil
larkspur
tulip
rose
orchid
gladiolus
lily
snapdragon
aster
protea
larkspur
lilac
iris
dahlia
*/
```

```
}
```

See also

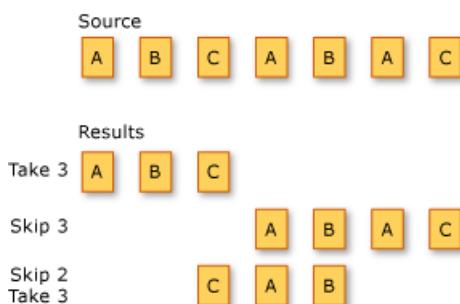
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [select clause](#)
- [How to populate object collections from multiple sources \(LINQ\) \(C#\)](#)
- [How to split a file into many files by using groups \(LINQ\) \(C#\)](#)

Partitioning Data (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Partitioning in LINQ refers to the operation of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections.

The following illustration shows the results of three different partitioning operations on a sequence of characters. The first operation returns the first three elements in the sequence. The second operation skips the first three elements and returns the remaining elements. The third operation skips the first two elements in the sequence and returns the next three elements.



The standard query operator methods that partition sequences are listed in the following section.

Operators

OPERATOR NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Skip	Skips elements up to a specified position in a sequence.	Not applicable.	Enumerable.Skip Queryable.Skip
SkipWhile	Skips elements based on a predicate function until an element does not satisfy the condition.	Not applicable.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Takes elements up to a specified position in a sequence.	Not applicable.	Enumerable.Take Queryable.Take
TakeWhile	Takes elements based on a predicate function until an element does not satisfy the condition.	Not applicable.	Enumerable.TakeWhile Queryable.TakeWhile

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)

Join Operations (C#)

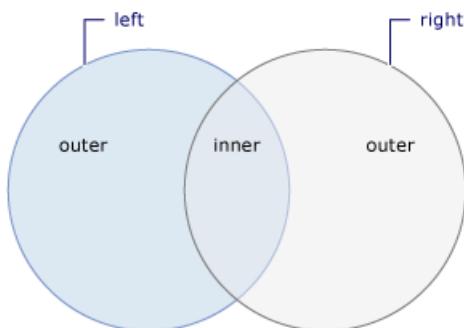
11/2/2020 • 4 minutes to read • [Edit Online](#)

A *join* of two data sources is the association of objects in one data source with objects that share a common attribute in another data source.

Joining is an important operation in queries that target data sources whose relationships to each other cannot be followed directly. In object-oriented programming, this could mean a correlation between objects that is not modeled, such as the backwards direction of a one-way relationship. An example of a one-way relationship is a Customer class that has a property of type City, but the City class does not have a property that is a collection of Customer objects. If you have a list of City objects and you want to find all the customers in each city, you could use a join operation to find them.

The join methods provided in the LINQ framework are [Join](#) and [GroupJoin](#). These methods perform equijoins, or joins that match two data sources based on equality of their keys. (For comparison, Transact-SQL supports join operators other than 'equals', for example the 'less than' operator.) In relational database terms, [Join](#) implements an inner join, a type of join in which only those objects that have a match in the other data set are returned. The [GroupJoin](#) method has no direct equivalent in relational database terms, but it implements a superset of inner joins and left outer joins. A left outer join is a join that returns each element of the first (left) data source, even if it has no correlated elements in the other data source.

The following illustration shows a conceptual view of two sets and the elements within those sets that are included in either an inner join or a left outer join.



Methods

Method Name	Description	C# Query Expression Syntax	More Information
Join	Joins two sequences based on key selector functions and extracts pairs of values.	<code>join ... in ... on ... equals ...</code>	Enumerable.Join Queryable.Join
GroupJoin	Joins two sequences based on key selector functions and groups the resulting matches for each element.	<code>join ... in ... on ... equals ... into ...</code>	Enumerable.GroupJoin Queryable.GroupJoin

Query expression syntax examples

Join

The following example uses the `join ... in ... on ... equals ...` clause to join two sequences based on specific value:

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
               join category in categories on product.CategoryId equals category.Id
               select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}

```

GroupJoin

The following example uses the `join ... in ... on ... equals ... into ...` clause to join two sequences based on specific value and groups the resulting matches for each element:

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //     Tea
    // Group
    //     Apple
    //     Kiwi
    // Group
    //     Carrot
}

```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Anonymous Types](#)
- [Formulate Joins and Cross-Product Queries](#)
- [join clause](#)

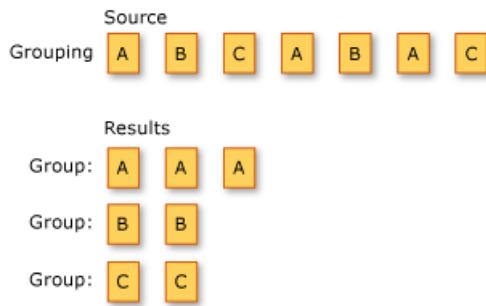
- [Join by using composite keys](#)
- [How to join content from dissimilar files \(LINQ\) \(C#\)](#)
- [Order the results of a join clause](#)
- [Perform custom join operations](#)
- [Perform grouped joins](#)
- [Perform inner joins](#)
- [Perform left outer joins](#)
- [How to populate object collections from multiple sources \(LINQ\) \(C#\)](#)

Grouping Data (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Grouping refers to the operation of putting data into groups so that the elements in each group share a common attribute.

The following illustration shows the results of grouping a sequence of characters. The key for each group is the character.



The standard query operator methods that group data elements are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
GroupBy	Groups elements that share a common attribute. Each group is represented by an <code>IGrouping< TKey, TElement ></code> object.	<code>group ... by</code> -or- <code>group ... by ... into ...</code>	Enumerable.GroupBy Queryable.GroupBy
ToLookup	Inserts elements into a <code>Lookup< TKey, TElement ></code> (a one-to-many dictionary) based on a key selector function.	Not applicable.	Enumerable.ToLookup

Query Expression Syntax Example

The following code example uses the `group by` clause to group integers in a list according to whether they are even or odd.

```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                             group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208
*/
```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [group clause](#)
- [Create a nested group](#)
- [How to group files by extension \(LINQ\) \(C#\)](#)
- [Group query results](#)
- [Perform a subquery on a grouping operation](#)
- [How to split a file into many files by using groups \(LINQ\) \(C#\)](#)

Generation Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Generation refers to creating a new sequence of values.

The standard query operator methods that perform generation are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
DefaultIfEmpty	Replaces an empty collection with a default valued singleton collection.	Not applicable.	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Empty	Returns an empty collection.	Not applicable.	Enumerable.Empty
Range	Generates a collection that contains a sequence of numbers.	Not applicable.	Enumerable.Range
Repeat	Generates a collection that contains one repeated value.	Not applicable.	Enumerable.Repeat

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)

Equality Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Two sequences whose corresponding elements are equal and which have the same number of elements are considered equal.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
SequenceEqual	Determines whether two sequences are equal by comparing elements in a pair-wise manner.	Not applicable.	Enumerable.SequenceEqual Queryable.SequenceEqual

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to compare the contents of two folders \(LINQ\) \(C#\)](#)

Element Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Element operations return a single, specific element from a sequence.

The standard query operator methods that perform element operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
ElementAt	Returns the element at a specified index in a collection.	Not applicable.	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Returns the element at a specified index in a collection or a default value if the index is out of range.	Not applicable.	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
First	Returns the first element of a collection, or the first element that satisfies a condition.	Not applicable.	Enumerable.First Queryable.First
FirstOrDefault	Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if no such element exists.	Not applicable.	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Last	Returns the last element of a collection, or the last element that satisfies a condition.	Not applicable.	Enumerable.Last Queryable.Last
LastOrDefault	Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.	Not applicable.	Enumerable.LastOrDefault Queryable.LastOrDefault
Single	Returns the only element of a collection or the only element that satisfies a condition. Throws an InvalidOperationException if there is no element or more than one element to return.	Not applicable.	Enumerable.Single Queryable.Single

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
SingleOrDefault	<p>Returns the only element of a collection or the only element that satisfies a condition. Returns a default value if there is no element to return. Throws an InvalidOperationException if there is more than one element to return.</p>	<p>Not applicable.</p>	<p>Enumerable.SingleOrDefault Queryable.SingleOrDefault</p>

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to query for the largest file or files in a directory tree \(LINQ\) \(C#\)](#)

Converting Data Types (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Conversion methods change the type of input objects.

Conversion operations in LINQ queries are useful in a variety of applications. Following are some examples:

- The [Enumerable.AsEnumerable](#) method can be used to hide a type's custom implementation of a standard query operator.
- The [Enumerable.OfType](#) method can be used to enable non-parameterized collections for LINQ querying.
- The [Enumerable.ToArray](#), [Enumerable.ToDictionary](#), [Enumerable.ToList](#), and [Enumerable.ToLookup](#) methods can be used to force immediate query execution instead of deferring it until the query is enumerated.

Methods

The following table lists the standard query operator methods that perform data-type conversions.

The conversion methods in this table whose names start with "As" change the static type of the source collection but do not enumerate it. The methods whose names start with "To" enumerate the source collection and put the items into the corresponding collection type.

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
AsEnumerable	Returns the input typed as IEnumerable<T> .	Not applicable.	Enumerable.AsEnumerable
AsQueryable	Converts a (generic) IEnumerable to a (generic) IQueryable .	Not applicable.	Queryable.AsQueryable
Cast	Casts the elements of a collection to a specified type.	Use an explicitly typed range variable. For example: <code>from string str in words</code>	Enumerable.Cast Queryable.Cast
OfType	Filters values, depending on their ability to be cast to a specified type.	Not applicable.	Enumerable.OfType Queryable.OfType
ToArray	Converts a collection to an array. This method forces query execution.	Not applicable.	Enumerable.ToArray
ToDictionary	Puts elements into a Dictionary< TKey, TValue > based on a key selector function. This method forces query execution.	Not applicable.	Enumerable.ToDictionary

Method Name	Description	C# Query Expression Syntax	More Information
ToList	Converts a collection to a <code>List<T></code> . This method forces query execution.	Not applicable.	Enumerable.ToList
ToLookup	Puts elements into a <code>Lookup< TKey, TElement ></code> (a one-to-many dictionary) based on a key selector function. This method forces query execution.	Not applicable.	Enumerable.ToLookup

Query Expression Syntax Example

The following code example uses an explicitly typed range variable to cast a type to a subtype before accessing a member that is available only on the subtype.

```

class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
               where cPlant.TrapType == "Snap Trap"
               select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:
       Venus Fly Trap
       Waterwheel Plant
    */
}

```

See also

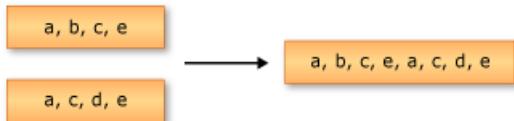
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [from clause](#)
- [LINQ Query Expressions](#)
- [How to query an ArrayList with LINQ \(C#\)](#)

Concatenation Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Concatenation refers to the operation of appending one sequence to another.

The following illustration depicts a concatenation operation on two sequences of characters.



The standard query operator methods that perform concatenation are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Concat	Concatenates two sequences to form one sequence.	Not applicable.	Enumerable.Concat Queryable.Concat

See also

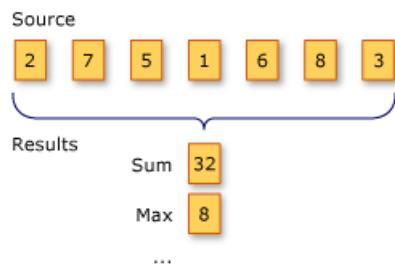
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to combine and compare string collections \(LINQ\) \(C#\)](#)

Aggregation Operations (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An aggregation operation computes a single value from a collection of values. An example of an aggregation operation is calculating the average daily temperature from a month's worth of daily temperature values.

The following illustration shows the results of two different aggregation operations on a sequence of numbers. The first operation sums the numbers. The second operation returns the maximum value in the sequence.



The standard query operator methods that perform aggregation operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Aggregate	Performs a custom aggregation operation on the values of a collection.	Not applicable.	Enumerable.Aggregate Queryable.Aggregate
Average	Calculates the average value of a collection of values.	Not applicable.	Enumerable.Average Queryable.Average
Count	Counts the elements in a collection, optionally only those elements that satisfy a predicate function.	Not applicable.	Enumerable.Count Queryable.Count
LongCount	Counts the elements in a large collection, optionally only those elements that satisfy a predicate function.	Not applicable.	Enumerable.LongCount Queryable.LongCount
Max	Determines the maximum value in a collection.	Not applicable.	Enumerable.Max Queryable.Max
Min	Determines the minimum value in a collection.	Not applicable.	Enumerable.Min Queryable.Min
Sum	Calculates the sum of the values in a collection.	Not applicable.	Enumerable.Sum Queryable.Sum

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to compute column values in a CSV text file \(LINQ\) \(C#\)](#)
- [How to query for the largest file or files in a directory tree \(LINQ\) \(C#\)](#)
- [How to query for the total number of bytes in a set of folders \(LINQ\) \(C#\)](#)

LINQ to Objects (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The term "LINQ to Objects" refers to the use of LINQ queries with any `IEnumerable` or `IEnumerable<T>` collection directly, without the use of an intermediate LINQ provider or API such as [LINQ to SQL](#) or [LINQ to XML](#). You can use LINQ to query any enumerable collections such as `List<T>`, `Array`, or `Dictionary< TKey, TValue >`. The collection may be user-defined or may be returned by a .NET API.

In a basic sense, LINQ to Objects represents a new approach to collections. In the old way, you had to write complex `foreach` loops that specified how to retrieve data from a collection. In the LINQ approach, you write declarative code that describes what you want to retrieve.

In addition, LINQ queries offer three main advantages over traditional `foreach` loops:

- They are more concise and readable, especially when filtering multiple conditions.
- They provide powerful filtering, ordering, and grouping capabilities with a minimum of application code.
- They can be ported to other data sources with little or no modification.

In general, the more complex the operation you want to perform on the data, the more benefit you'll realize by using LINQ instead of traditional iteration techniques.

The purpose of this section is to demonstrate the LINQ approach with some select examples. It's not intended to be exhaustive.

In This Section

[LINQ and Strings \(C#\)](#)

Explains how LINQ can be used to query and transform strings and collections of strings. Also includes links to articles that demonstrate these principles.

[LINQ and Reflection \(C#\)](#)

Links to a sample that demonstrates how LINQ uses reflection.

[LINQ and File Directories \(C#\)](#)

Explains how LINQ can be used to interact with file systems. Also includes links to articles that demonstrate these concepts.

[How to query an ArrayList with LINQ \(C#\)](#)

Demonstrates how to query an `ArrayList` in C#.

[How to add custom methods for LINQ queries \(C#\)](#)

Explains how to extend the set of methods that you can use for LINQ queries by adding extension methods to the `IEnumerable<T>` interface.

[Language-Integrated Query \(LINQ\) \(C#\)](#)

Provides links to articles that explain LINQ and provide examples of code that perform queries.

LINQ and strings (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

LINQ can be used to query and transform strings and collections of strings. It can be especially useful with semi-structured data in text files. LINQ queries can be combined with traditional string functions and regular expressions. For example, you can use the [String.Split](#) or [Regex.Split](#) method to create an array of strings that you can then query or modify by using LINQ. You can use the [Regex.IsMatch](#) method in the `where` clause of a LINQ query. And you can use LINQ to query or modify the [MatchCollection](#) results returned by a regular expression.

You can also use the techniques described in this section to transform semi-structured text data to XML. For more information, see [How to generate XML from CSV files](#).

The examples in this section fall into two categories:

Querying a block of text

You can query, analyze, and modify text blocks by splitting them into a queryable array of smaller strings by using the [String.Split](#) method or the [Regex.Split](#) method. You can split the source text into words, sentences, paragraphs, pages, or any other criteria, and then perform additional splits if they are required in your query.

- [How to count occurrences of a word in a string \(LINQ\) \(C#\)](#)
Shows how to use LINQ for simple querying over text.
- [How to query for sentences that contain a specified set of words \(LINQ\) \(C#\)](#)
Shows how to split text files on arbitrary boundaries and how to perform queries against each part.
- [How to query for characters in a string \(LINQ\) \(C#\)](#)
Demonstrates that a string is a queryable type.
- [How to combine LINQ queries with regular expressions \(C#\)](#)
Shows how to use regular expressions in LINQ queries for complex pattern matching on filtered query results.

Querying semi-structured data in text format

Many different types of text files consist of a series of lines, often with similar formatting, such as tab- or comma-delimited files or fixed-length lines. After you read such a text file into memory, you can use LINQ to query and/or modify the lines. LINQ queries also simplify the task of combining data from multiple sources.

- [How to find the set difference between two lists \(LINQ\) \(C#\)](#)
Shows how to find all the strings that are present in one list but not the other.
- [How to sort or filter text data by any word or field \(LINQ\) \(C#\)](#)
Shows how to sort text lines based on any word or field.
- [How to reorder the fields of a delimited file \(LINQ\) \(C#\)](#)
Shows how to reorder fields in a line in a .csv file.
- [How to combine and compare string collections \(LINQ\) \(C#\)](#)

Shows how to combine string lists in various ways.

- [How to populate object collections from multiple sources \(LINQ\) \(C#\)](#)

Shows how to create object collections by using multiple text files as data sources.

- [How to join content from dissimilar files \(LINQ\) \(C#\)](#)

Shows how to combine strings in two lists into a single string by using a matching key.

- [How to split a file into many files by using groups \(LINQ\) \(C#\)](#)

Shows how to create new files by using a single file as a data source.

- [How to compute column values in a CSV text file \(LINQ\) \(C#\)](#)

Shows how to perform mathematical computations on text data in .csv files.

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [How to generate XML from CSV files](#)

How to count occurrences of a word in a string (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to use a LINQ query to count the occurrences of a specified word in a string. Note that to perform the count, first the [Split](#) method is called to create an array of words. There is a performance cost to the [Split](#) method. If the only operation on the string is to count the words, you should consider using the [Matches](#) or [IndexOf](#) methods instead. However, if performance is not a critical issue, or you have already split the sentence in order to perform other types of queries over it, then it makes sense to use LINQ to count the words or phrases as well.

Example

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects" +
            @" have not been well integrated. Programmers work in C# or Visual Basic" +
            @" and also in SQL or XQuery. On the one side are concepts such as classes," +
            @" objects, fields, inheritance, and .NET APIs. On the other side" +
            @" are tables, columns, rows, nodes, and separate languages for dealing with" +
            @" them. Data types often require translation between the two worlds; there are" +
            @" different standard functions. Because the object world has no notion of query, a" +
            @" query can only be represented as a string without compile-time type checking or" +
            @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to" +
            @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';' }, StringSplitOptions.RemoveEmptyEntries);

        // Create the query. Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                        where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                        select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrence(s) of the search term \"{1}\" were found.", wordCount,
            searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to query for sentences that contain a specified set of words (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to find sentences in a text file that contain matches for each of a specified set of words. Although the array of search terms is hard-coded in this example, it could also be populated dynamically at runtime. In this example, the query returns the sentences that contain the words "Historically," "data," and "integrated."

Example

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects " +
            @"have not been well integrated. Programmers work in C# or Visual Basic " +
            @"and also in SQL or XQuery. On the one side are concepts such as classes, " +
            @"objects, fields, inheritance, and .NET APIs. On the other side " +
            @"are tables, columns, rows, nodes, and separate languages for dealing with " +
            @"them. Data types often require translation between the two worlds; there are " +
            @"different standard functions. Because the object world has no notion of query, a " +
            @"query can only be represented as a string without compile-time type checking or " +
            @"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
            @"objects in memory is often tedious and error-prone.';

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at runtime.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch array.
        // Note that the number of terms to match is not specified at compile time.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';' , ':' , ',' }, StringSplitOptions.RemoveEmptyEntries)
                            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

The query works by first splitting the text into sentences, and then splitting the sentences into an array of strings

that hold each word. For each of these arrays, the `Distinct` method removes all duplicate words, and then the query performs an `Intersect` operation on the word array and the `wordsToMatch` array. If the count of the intersection is the same as the count of the `wordsToMatch` array, all words were found in the words and the original sentence is returned.

In the call to `Split`, the punctuation marks are used as separators in order to remove them from the string. If you did not do this, for example you could have a string "Historically," that would not match "Historically" in the `wordsToMatch` array. You may have to use additional separators, depending on the types of punctuation found in the source text.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to query for characters in a string (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Because the `String` class implements the generic `IEnumerable<T>` interface, any string can be queried as a sequence of characters. However, this is not a common use of LINQ. For complex pattern matching operations, use the `Regex` class.

Example

The following example queries a string to determine the number of numeric digits it contains. Note that the query is "reused" after it is executed the first time. This is possible because the query itself does not store any actual results.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
            from ch in aString
            where Char.IsDigit(ch)
            select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Output: 9 9 7 4 1 2 8 9
Count = 8
ABCDE99F
*/
```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [How to combine LINQ queries with regular expressions \(C#\)](#)

How to combine LINQ queries with regular expressions (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to use the `Regex` class to create a regular expression for more complex matching in text strings. The LINQ query makes it easy to filter on exactly the files that you want to search with the regular expression, and to shape the results.

Example

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match match in matches
                                select match.Value
            };
        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
            foreach (var v2 in v.matchedValues)
            {

```

```

        Console.WriteLine(" " + v2);
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}

// This method assumes that the application has discovery
// permissions for all folders under the specified path.
static IEnumerable<System.IO.FileInfo> GetFiles(string path)
{
    if (!System.IO.Directory.Exists(path))
        throw new System.IO.DirectoryNotFoundException();

    string[] fileNames = null;
    List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

    fileNames = System.IO.Directory.GetFiles(path, "*.*", System.IO.SearchOption.AllDirectories);
    foreach (string name in fileNames)
    {
        files.Add(new System.IO.FileInfo(name));
    }
    return files;
}
}

```

Note that you can also query the [MatchCollection](#) object that is returned by a `RegEx` search. In this example only the value of each match is produced in the results. However, it is also possible to use LINQ to perform all kinds of filtering, sorting, and grouping on that collection. Because [MatchCollection](#) is a non-generic `IEnumerable` collection, you have to explicitly state the type of the range variable in the query.

Compiling the Code

Create a C# console application project with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to find the set difference between two lists (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to use LINQ to compare two lists of strings and output those lines that are in names1.txt but not in names2.txt.

To create the data files

1. Copy names1.txt and names2.txt to your solution folder as shown in [How to combine and compare string collections \(LINQ\) \(C#\)](#).

Example

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
            names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   The following lines are in names1.txt but not names2.txt
   Potra, Cristina
   Noriega, Fabricio
   Aw, Kam Foo
   Toyoshima, Tim
   Guy, Wey Yuan
   Garcia, Debra
*/
```

Some types of query operations in C#, such as [Except](#), [Distinct](#), [Union](#), and [Concat](#), can only be expressed in method-based syntax.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to sort or filter text data by any word or field (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following example shows how to sort lines of structured text, such as comma-separated values, by any field in the line. The field may be dynamically specified at runtime. Assume that the fields in scores.csv represent a student's ID number, followed by a series of four test scores.

To create a file that contains data

1. Copy the scores.csv data from the topic [How to join content from dissimilar files \(LINQ\) \(C#\)](#) and save it to your solution folder.

Example

```

public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}

/* Output (if sortField == 1):
Sorted highest to lowest by field [1]:
116, 99, 86, 90, 94
120, 99, 82, 81, 79
111, 97, 92, 81, 60
114, 97, 89, 85, 82
121, 96, 85, 91, 60
122, 94, 92, 91, 91
117, 93, 92, 80, 87
118, 92, 90, 83, 78
113, 88, 94, 65, 91
112, 75, 84, 91, 39
119, 68, 79, 88, 92
115, 35, 72, 91, 70
*/

```

This example also demonstrates how to return a query variable from a method.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to reorder the fields of a delimited file (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A comma-separated value (CSV) file is a text file that is often used to store spreadsheet data or other tabular data that is represented by rows and columns. By using the [Split](#) method to separate the fields, it is very easy to query and manipulate CSV files by using LINQ. In fact, the same technique can be used to reorder the parts of any structured line of text; it is not limited to CSV files.

In the following example, assume that the three columns represent students' "last name," "first name", and "ID." The fields are in alphabetical order based on the students' last names. The query produces a new sequence in which the ID column appears first, followed by a second column that combines the student's first name and last name. The lines are reordered according to the ID field. The results are saved into a new file and the original data is not modified.

To create the data file

1. Copy the following lines into a plain text file that is named `spreadsheet1.csv`. Save the file in your project folder.

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

Example

```

class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.File.ReadAllLines(@"../../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/

```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)
- [How to generate XML from CSV files \(C#\)](#)

How to combine and compare string collections (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to merge files that contain lines of text and then sort the results. Specifically, it shows how to perform a simple concatenation, a union, and an intersection on the two sets of text lines.

To set up the project and the text files

1. Copy these names into a text file that is named names1.txt and save it in your project folder:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copy these names into a text file that is named names2.txt and save it in your project folder. Note that the two files have some names in common.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Example

```
class MergeStrings
{
    static void Main(string[] args)
    {
        //Put text files in your solution folder
        string[] fileA = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../names2.txt");

        //Simple concatenation and sort. Duplicates are preserved.
        IEnumerable<string> concatQuery =
            fileA.Concat(fileB).OrderBy(s => s);

        // Pass the query variable to another function for execution.
        OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are preserved:");

        // Concatenate and remove duplicate names based on
        // default string comparer.
        IEnumerable<string> uniqueNamesQuery =
```

```

        fileA.Union(fileB).OrderBy(s => s);
        OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

        // Find the names that occur in both files (based on
        // default string comparer).
        IEnumerable<string> commonNamesQuery =
            fileA.Intersect(fileB);
        OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

        // Find the matching fields in each list. Merge the two
        // results by using Concat, and then
        // sort using the default string comparer.
        string nameMatch = "Garcia";

        IEnumerable<String> tempQuery1 =
            from name in fileA
            let n = name.Split(',')
            where n[0] == nameMatch
            select name;

        IEnumerable<string> tempQuery2 =
            from name2 in fileB
            let n2 = name2.Split(',')
            where n2[0] == nameMatch
            select name2;

        IEnumerable<string> nameMatchQuery =
            tempQuery1.Concat(tempQuery2).OrderBy(s => s);
        OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \'{nameMatch}\':");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
   Simple concatenate and sort. Duplicates are preserved:
   Aw, Kam Foo
   Bankov, Peter
   Bankov, Peter
   Beebe, Ann
   Beebe, Ann
   El Yassir, Mehdi
   Garcia, Debra
   Garcia, Hugo
   Garcia, Hugo
   Giakoumakis, Leo
   Gilchrist, Beth
   Guy, Wey Yuan
   Holm, Michael
   Holm, Michael
   Liu, Jinghao
   McLin, Nkenge
   Myrcha, Jacek
   Noriega, Fabricio
   Potra, Cristina
   Toyoshima, Tim
   20 total names in list

```

```
Union removes duplicate names:
```

```
Aw, Kam Foo
Bankov, Peter
Beebe, Ann
El Yassir, Mehdi
Garcia, Debra
Garcia, Hugo
Giakoumakis, Leo
Gilchrist, Beth
Guy, Wey Yuan
Holm, Michael
Liu, Jinghao
McLin, Nkenge
Myrcha, Jacek
Noriega, Fabricio
Potra, Cristina
Toyoshima, Tim
16 total names in list
```

```
Merge based on intersect:
```

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
4 total names in list
```

```
Concat based on partial name match "Garcia":
```

```
Garcia, Debra
Garcia, Hugo
Garcia, Hugo
3 total names in list
```

```
*/
```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to populate object collections from multiple sources (LINQ) (C#)

11/2/2020 • 4 minutes to read • [Edit Online](#)

This example shows how to merge data from different sources into a sequence of new types.

NOTE

Don't try to join in-memory data or data in the file system with data that is still in a database. Such cross-domain joins can yield undefined results because of different ways in which join operations might be defined for database queries and other types of sources. Additionally, there is a risk that such an operation could cause an out-of-memory exception if the amount of data in the database is large enough. To join data from a database to in-memory data, first call `ToList` or `ToArray` on the database query, and then perform the join on the returned collection.

To create the data file

Copy the names.csv and scores.csv files into your project folder, as described in [How to join content from dissimilar files \(LINQ\) \(C#\)](#).

Example

The following example shows how to use a named type `Student` to store merged data from two in-memory collections of strings that simulate spreadsheet data in .csv format. The first collection of strings represents the student names and IDs, and the second collection represents the student ID (in the first column) and four exam scores. The ID is used as the foreign key.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and an
        // ID number, separated by commas. For example, Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
    }
}
```

```


    // var could be used instead of an explicit type. Note the dynamic
    // creation of a list of ints for the ExamScores member. The first item
    // is skipped in the split string because it is the student ID,
    // not an exam score.
    IEnumerable<Student> queryNamesScores =
        from nameLine in names
        let splitName = nameLine.Split(',')
        from scoreLine in scores
        let splitScoreLine = scoreLine.Split(',')
        where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
        select new Student()
    {
        FirstName = splitName[0],
        LastName = splitName[1],
        ID = Convert.ToInt32(splitName[2]),
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText)).
                      ToList()
    };
}

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.FirstName, student.LastName,
        student.ExamScores.Average());
}

//Keep console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/


```

In the `select` clause, an object initializer is used to instantiate each new `Student` object by using the data from the two sources.

If you don't have to store the results of a query, anonymous types can be more convenient than named types. Named types are required if you pass the query results outside the method in which the query is executed. The following example executes the same task as the previous example, but uses anonymous types instead of named types:

```
// Merge the data sources by using an anonymous type.  
// Note the dynamic creation of a list of ints for the  
// ExamScores member. We skip 1 because the first string  
// in the array is the student ID, not an exam score.  
var queryNamesScores2 =  
    from nameLine in names  
    let splitName = nameLine.Split(',')  
    from scoreLine in scores  
    let splitScoreLine = scoreLine.Split(',')  
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])  
    select new  
    {  
        First = splitName[0],  
        Last = splitName[1],  
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)  
                      select Convert.ToInt32(scoreAsText))  
                      .ToList()  
    };  
  
// Display each student's name and exam score average.  
foreach (var student in queryNamesScores2)  
{  
    Console.WriteLine("The average score of {0} {1} is {2}.",  
        student.First, student.Last, student.ExamScores.Average());  
}
```

See also

- [LINQ and Strings \(C#\)](#)
- [Object and Collection Initializers](#)
- [Anonymous Types](#)

How to split a file into many files by using groups (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows one way to merge the contents of two files and then create a set of new files that organize the data in a new way.

To create the data files

1. Copy these names into a text file that is named names1.txt and save it in your project folder:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copy these names into a text file that is named names2.txt and save it in your project folder: Note that the two files have some names in common.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Example

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                         let n = name.Split(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;
```

```

// Create a new file for each group that was created
// Note that nested foreach loops are required to access
// individual items with each group.
foreach (var g in groupQuery)
{
    // Create the new file name.
    string fileName = @"../../testFile_" + g.Key + ".txt";

    // Output to display.
    Console.WriteLine(g.Key);

    // Write file.
    using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
    {
        foreach (var item in g)
        {
            sw.WriteLine(item);
            // Output to console for example purposes.
            Console.WriteLine("  {0}", item);
        }
    }
}

// Keep console window open in debug mode.
Console.WriteLine("Files have been written. Press any key to exit");
Console.ReadKey();
}

/* Output:
A
Aw, Kam Foo
B
Bankov, Peter
Beebe, Ann
E
El Yassir, Mehdi
G
Garcia, Hugo
Guy, Wey Yuan
Garcia, Debra
Gilchrist, Beth
Giakoumakis, Leo
H
Holm, Michael
L
Liu, Jinghao
M
Myrcha, Jacek
McLin, Nkenge
N
Noriega, Fabricio
P
Potra, Cristina
T
Toyoshima, Tim
*/

```

The program writes a separate file for each group in the same folder as the data files.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)

- LINQ and File Directories (C#)

How to join content from dissimilar files (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to join data from two comma-delimited files that share a common value that is used as a matching key. This technique can be useful if you have to combine data from two spreadsheets, or from a spreadsheet and from a file that has another format, into a new file. You can modify the example to work with any kind of structured text.

To create the data files

1. Copy the following lines into a file that is named *scores.csv* and save it to your project folder. The file represents spreadsheet data. Column 1 is the student's ID, and columns 2 through 5 are test scores.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

2. Copy the following lines into a file that is named *names.csv* and save it to your project folder. The file represents a spreadsheet that contains the student's last name, first name, and student ID.

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Example

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
```

```

// and a set of four test scores. The following query joins
// the scores to the student names by using ID as a
// matching key.

string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

// Name:    Last[0],      First[1],   ID[2]
//          Omelchenko,   Svetlana,   11
// Score:   StudentID[0], Exam1[1]  Exam2[2],  Exam3[3],  Exam4[4]
//          111,           97,        92,       81,       60

// This query joins two dissimilar spreadsheets based on common ID value.
// Multiple from clauses are used instead of a join clause
// in order to store results of id.Split.
IEnumerable<string> scoreQuery1 =
    from name in names
    let nameFields = name.Split(',')
    from id in scores
    let scoreFields = id.Split(',')
    where Convert.ToInt32(nameFields[2]) == Convert.ToInt32(scoreFields[0])
    select nameFields[0] + "," + scoreFields[1] + "," + scoreFields[2]
        + "," + scoreFields[3] + "," + scoreFields[4];

// Pass a query variable to a method and execute it
// in the method. The query itself is unchanged.
OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
}

/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
*/

```

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to compute column values in a CSV text file (LINQ) (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

This example shows how to perform aggregate computations such as Sum, Average, Min, and Max on the columns of a .csv file. The example principles that are shown here can be applied to other types of structured text.

To create the source file

1. Copy the following lines into a file that is named scores.csv and save it in your project folder. Assume that the first column represents a student ID, and subsequent columns represent scores from four exams.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Example

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../../../../scores.csv");

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID      Exam#1  Exam#2  Exam#3  Exam#4
        // 111,           97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
```

```

// run the calculations on. This value could be
// passed in dynamically at runtime.

// Variable columnQuery is an IEnumerable<int>.
// The following query performs two steps:
// 1) use Split to break each row (a string) into an array
//     of strings,
// 2) convert the element at position examNum to an int
//     and select it.
var columnQuery =
    from line in strs
    let elements = line.Split(',')
    select Convert.ToInt32(elements[examNum]);

// Execute the query and cache the results to improve
// performance. This is helpful only with very large files.
var results = columnQuery.ToList();

// Perform aggregate calculations Average, Max, and
// Min on the column specified by examNum.
double average = results.Average();
int max = results.Max();
int min = results.Min();

Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
    examNum, average, max, min);
}

static void MultiColumns(IEnumerable<string> strs)
{
    Console.WriteLine("Multi Column Query:");

    // Create a query, multiColQuery. Explicit typing is used
    // to make clear that, when executed, multiColQuery produces
    // nested sequences. However, you get the same results by
    // using 'var'.

    // The multiColQuery query performs the following steps:
    // 1) use Split to break each row (a string) into an array
    //     of strings,
    // 2) use Skip to skip the "Student ID" column, and store the
    //     rest of the row in scores.
    // 3) convert each score in the current row from a string to
    //     an int, and select that entire sequence as one row
    //     in the results.
    I Enumerable<I Enumerable<int>> multiColQuery =
        from line in strs
        let elements = line.Split(',')
        let scores = elements.Skip(1)
        select (from str in scores
            select Convert.ToInt32(str));

    // Execute the query and cache the results to improve
    // performance.
    // ToArray could be used instead ofToList.
    var results = multiColQuery.ToList();

    // Find out how many columns you have in results.
    int columnCount = results[0].Count();

    // Perform aggregate calculations Average, Max, and
    // Min on each column.
    // Perform one iteration of the loop for each column
    // of scores.
    // You can use a for loop instead of a foreach loop
    // because you already executed the multiColQuery
    // query by calling ToList.
    for (int column = 0; column < columnCount; column++)
    {

```

```

        var results2 = from row in results
                      select row.ElementAt(column);
        double average = results2.Average();
        int max = results2.Max();
        int min = results2.Min();

        // Add one to column because the first exam is Exam #1,
        // not Exam #0.
        Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                          column + 1, average, max, min);
    }
}
/* Output:
Single Column Query:
Exam #4: Average:76.92 High Score:94 Low Score:39

Multi Column Query:
Exam #1 Average: 86.08 High Score: 99 Low Score: 35
Exam #2 Average: 86.42 High Score: 94 Low Score: 72
Exam #3 Average: 84.75 High Score: 91 Low Score: 65
Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/

```

The query works by using the [Split](#) method to convert each line of text into an array. Each array element represents a column. Finally, the text in each column is converted to its numeric representation. If your file is a tab-separated file, just update the argument in the `Split` method to `\t`.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query an assembly's metadata with Reflection (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The .NET reflection APIs can be used to examine the metadata in a .NET assembly and create collections of types, type members, parameters, and so on that are in that assembly. Because these collections support the generic `IEnumerable<T>` interface, they can be queried by using LINQ.

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

Example

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                            where type.IsPublic
                            from method in type.GetMethods()
                            where method.ReturnType.IsArray == true
                                || ( method.ReturnType.GetInterface(
                                    typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                                && method.ReturnType.FullName != "System.String" )
                            group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine(" {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

The example uses the `Assembly.GetTypes` method to return an array of types in the specified assembly. The `where` filter is applied so that only public types are returned. For each public type, a subquery is generated by using the `MethodInfo` array that is returned from the `Type.GetMethods` call. These results are filtered to return only those methods whose return type is an array or else a type that implements `IEnumerable<T>`. Finally, these results are grouped by using the type name as a key.

See also

- LINQ to Objects (C#)

How to query an assembly's metadata with Reflection (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The .NET reflection APIs can be used to examine the metadata in a .NET assembly and create collections of types, type members, parameters, and so on that are in that assembly. Because these collections support the generic `IEnumerable<T>` interface, they can be queried by using LINQ.

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

Example

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                            where type.IsPublic
                            from method in type.GetMethods()
                            where method.ReturnType.IsArray == true
                                || ( method.ReturnType.GetInterface(
                                    typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                                && method.ReturnType.FullName != "System.String" )
                            group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine(" {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

The example uses the `Assembly.GetTypes` method to return an array of types in the specified assembly. The `where` filter is applied so that only public types are returned. For each public type, a subquery is generated by using the `MethodInfo` array that is returned from the `Type.GetMethods` call. These results are filtered to return only those methods whose return type is an array or else a type that implements `IEnumerable<T>`. Finally, these results are grouped by using the type name as a key.

See also

- LINQ to Objects (C#)

LINQ and file directories (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Many file system operations are essentially queries and are therefore well suited to the LINQ approach.

The queries in this section are non-destructive. They are not used to change the contents of the original files or folders. This follows the rule that queries should not cause any side-effects. In general, any code (including queries that perform create / update / delete operators) that modifies source data should be kept separate from the code that just queries the data.

This section contains the following topics:

[How to query for files with a specified attribute or name \(C#\)](#)

Shows how to search for files by examining one or more properties of its [FileInfo](#) object.

[How to group files by extension \(LINQ\) \(C#\)](#)

Shows how to return groups of [FileInfo](#) object based on their file name extension.

[How to query for the total number of bytes in a set of folders \(LINQ\) \(C#\)](#)

Shows how to return the total number of bytes in all the files in a specified directory tree.

[How to compare the contents of two folders \(LINQ\) \(C#\)](#)

Shows how to return all the files that are present in two specified folders, and also all the files that are present in one folder but not the other.

[How to query for the largest file or files in a directory tree \(LINQ\) \(C#\)](#)

Shows how to return the largest or smallest file, or a specified number of files, in a directory tree.

[How to query for duplicate files in a directory tree \(LINQ\) \(C#\)](#)

Shows how to group for all file names that occur in more than one location in a specified directory tree. Also shows how to perform more complex comparisons based on a custom comparer.

[How to query the contents of files in a folder \(LINQ\) \(C#\)](#)

Shows how to iterate through folders in a tree, open each file, and query the file's contents.

Comments

There is some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of [FileInfo](#) objects that represents all the files under a specified root folder and all its subfolders. The actual state of each [FileInfo](#) may change in the time between when you begin and end executing a query. For example, you can create a list of [FileInfo](#) objects to use as a data source. If you try to access the `Length` property in a query, the [FileInfo](#) object will try to access the file system to update the value of `Length`. If the file no longer exists, you will get a [FileNotFoundException](#) in your query, even though you are not querying the file system directly. Some queries in this section use a separate method that consumes these particular exceptions in certain cases. Another option is to keep your data source updated dynamically by using the [FileSystemWatcher](#).

See also

- [LINQ to Objects \(C#\)](#)

How to query for files with a specified attribute or name (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to find all files that have a specified file name extension (for example ".txt") in a specified directory tree. It also shows how to return either the newest or oldest file in the tree based on the creation time.

Example

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
            orderby file.CreationTime
            select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to group files by extension (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how LINQ can be used to perform advanced grouping and sorting operations on lists of files or folders. It also shows how to page output in the console window by using the `Skip` and `Take` methods.

Example

The following query shows how to group the contents of a specified directory tree by the file name extension.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    // and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles(".*",
System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
        IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            Console.WriteLine(filegroup.Key);
            foreach (var file in filegroup)
                Console.WriteLine(file.FullName);
            if (Console.ReadKey(true).Key != ConsoleKey.Spacebar)
                goAgain = false;
        }
    }
}
```

```
int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}

}
```

The output from this program can be long, depending on the details of the local file system and what the `startFolder` is set to. To enable viewing of all results, this example shows how to page through results. The same techniques can be applied to Windows and Web applications. Notice that because the code pages the items in a group, a nested `foreach` loop is required. There is also some additional logic to compute the current position in the list, and to enable the user to stop paging and exit the program. In this particular case, the paging query is run against the cached results from the original query. In other contexts, such as LINQ to SQL, such caching is not required.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- LINQ to Objects (C#)
 - LINQ and File Directories (C#)

How to query for the total number of bytes in a set of folders (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to retrieve the total number of bytes used by all the files in a specified folder and all its subfolders.

Example

The [Sum](#) method adds the values of all the items selected in the `select` clause. You can easily modify this query to retrieve the biggest or smallest file in the specified directory tree by calling the [Min](#) or [Max](#) method instead of [Sum](#).

```

class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}

```

If you only have to count the number of bytes in a specified directory tree, you can do this more efficiently without creating a LINQ query, which incurs the overhead of creating the list collection as a data source. The usefulness of the LINQ approach increases as the query becomes more complex, or when you have to run multiple queries against the same data source.

The query calls out to a separate method to obtain the file length. It does this in order to consume the possible exception that will be raised if the file was deleted on another thread after the [FileInfo](#) object was created in the call to [GetFiles](#). Even though the [FileInfo](#) object has already been created, the exception can occur because a [FileInfo](#) object will try to refresh its [Length](#) property with the most current length the first time the property is accessed. By putting this operation in a try-catch block outside the query, the code follows the rule of avoiding operations in queries that can cause side-effects. In general, great care must be taken when you consume

exceptions to make sure that an application is not left in an unknown state.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to compare the contents of two folders (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example demonstrates three ways to compare two file listings:

- By querying for a Boolean value that specifies whether the two file lists are identical.
- By querying for the intersection to retrieve the files that are in both folders.
- By querying for the set difference to retrieve the files that are in one folder but not the other.

NOTE

The techniques shown here can be adapted to compare sequences of objects of any type.

The `FileComparer` class shown here demonstrates how to use a custom comparer class together with the Standard Query Operators. The class is not intended for use in real-world scenarios. It just uses the name and length in bytes of each file to determine whether the contents of each folder are identical or not. In a real-world scenario, you should modify this comparer to perform a more rigorous equality check.

Example

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
        }
    }
}
```

```

        Console.WriteLine("The two folders are the same.");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine("The following files are in both folders:");
        foreach (var v in queryCommonFiles)
        {
            Console.WriteLine(v.FullName); //shows which items end up in result list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }

    // Find the set difference between the two folders.
    // For this example we only check one way.
    var queryList1Only = (from file in list1
                           select file).Except(list2, myFileCompare);

    Console.WriteLine("The following files are in list1 but not list2:");
    foreach (var v in queryList1Only)
    {
        Console.WriteLine(v.FullName);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
{
    public FileCompare() { }

    public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
    {
        return (f1.Name == f2.Name &&
               f1.Length == f2.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
    // also be equal. Because equality as defined here is a simple value equality, not
    // reference identity, it is possible that two or more objects will produce the same
    // hash code.
    public int GetHashCode(System.IO.FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}
}

```

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query for the largest file or files in a directory tree (LINQ) (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

This example shows five queries related to file size in bytes:

- How to retrieve the size in bytes of the largest file.
- How to retrieve the size in bytes of the smallest file.
- How to retrieve the [FileInfo](#) object largest or smallest file from one or more folders under a specified root folder.
- How to retrieve a sequence such as the 10 largest files.
- How to order files into groups based on their file size in bytes, ignoring files that are less than a specified size.

Example

The following example contains five separate queries that show how to query and group files, depending on their file size in bytes. You can easily modify these examples to base the query on some other property of the [FileInfo](#) object.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
            .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
            ...
```

```

(from file in fileList
    let len = GetFileLength(file)
    where len > 0
    orderby len descending
    select file)
.Fist();

Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                  startFolder, longestFile.FullName, longestFile.Length);

//Return the FileInfo of the smallest file
System.IO.FileInfo smallestFile =
    (from file in fileList
        let len = GetFileLength(file)
        where len > 0
        orderby len ascending
        select file).First();

Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                  startFolder, smallestFile.FullName, smallestFile.Length);

//Return the FileInfos for the 10 largest files
// queryTenLargest is an IEnumerable<System.IO.FileInfo>
var queryTenLargest =
    (from file in fileList
        let len = GetFileLength(file)
        orderby len descending
        select file).Take(10);

Console.WriteLine("The 10 largest files under {0} are:", startFolder);

foreach (var v in queryTenLargest)
{
    Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups =
    from file in fileList
    let len = GetFileLength(file)
    where len > 0
    group file by (len / 100000) into fileGroup
    where fileGroup.Key >= 2
    orderby fileGroup.Key descending
    select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine(filegroup.Key.ToString() + "0000");
    foreach (var item in filegroup)
    {
        Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
    }
}
}

// This method is used to swallow the possible exception
// that can be raised when accessing the FileInfo.Length property.
// In this particular case, it is safe to swallow the exception.
static long GetFileLength(System.IO.FileInfo fi)
{
    long retval;
    try
    {
        retval = fi.Length;
    }
    catch (System.IO.FileNotFoundException)
    {

```

```
// If a file is no longer present,  
// just add zero bytes to the total.  
retval = 0;  
}  
return retval;  
}  
}
```

To return one or more complete [FileInfo](#) objects, the query first must examine each one in the data source, and then sort them by the value of their [Length](#) property. Then it can return the single one or the sequence with the greatest lengths. Use [First](#) to return the first element in a list. Use [Take](#) to return the first n number of elements. Specify a descending sort order to put the smallest elements at the start of the list.

The query calls out to a separate method to obtain the file size in bytes in order to consume the possible exception that will be raised in the case where a file was deleted on another thread in the time period since the [FileInfo](#) object was created in the call to [GetFiles](#). Even though the [FileInfo](#) object has already been created, the exception can occur because a [FileInfo](#) object will try to refresh its [Length](#) property by using the most current size in bytes the first time the property is accessed. By putting this operation in a try-catch block outside the query, we follow the rule of avoiding operations in queries that can cause side-effects. In general, great care must be taken when consuming exceptions, to make sure that an application is not left in an unknown state.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query for duplicate files in a directory tree (LINQ) (C#)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Sometimes files that have the same name may be located in more than one folder. For example, under the Visual Studio installation folder, several folders have a readme.htm file. This example shows how to query for such duplicate file names under a specified root folder. The second example shows how to query for files whose size and LastWrite times also match.

Example

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
    }
}
```

```

public long Length { get; set; }

public override bool Equals(object obj)
{
    PortableKey other = (PortableKey)obj;
    return other.LastWriteTime == this.LastWriteTime &&
           other.Length == this.Length &&
           other.Name == this.Name;
}

public override int GetHashCode()
{
    string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
    return str.GetHashCode();
}

public override string ToString()
{
    return $"{this.Name} {this.Length} {this.LastWriteTime}";
}

static void QueryDuplicates2()
{
    // Change the root drive or folder if necessary.
    string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
    IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
        System.IO.SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //

    var queryDupFiles =
        from file in fileList
        group file.FullName.Substring(charsToSkip) by
            new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length
    } into fileGroup
        where fileGroup.Count() > 1
        select fileGroup;

    var list = queryDupFiles.ToList();

    int i = queryDupFiles.Count();

    PageOutput<PortableKey, string>(queryDupFiles);
}

// A generic method to page the output of the QueryDuplications methods
// Here the type of the group must be specified explicitly. "var" cannot
// be used in method signatures. This method does not display more than one
// group per page.
private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
{
    // Flag to break out of paging loop.
    bool goAgain = true;

    // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
    int numLines = Console.WindowHeight - 3;

    // Iterate through the outer collection of groups.
    foreach (var filegroup in groupByExtList)
    {

```

```

// Start a new extension at the top of a page.
int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var fileName in resultPage)
    {
        Console.WriteLine("\t{0}", fileName);
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

The first query uses a simple key to determine a match; this finds files that have the same name but whose contents might be different. The second query uses a compound key to match against three properties of the [FileInfo](#) object. This query is much more likely to find files that have the same name and similar or identical content.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query the contents of text files in a folder (LINQ) (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example shows how to query over all the files in a specified directory tree, open each file, and inspect its contents. This type of technique could be used to create indexes or reverse indexes of the contents of a directory tree. A simple string search is performed in this example. However, more complex types of pattern matching can be performed with a regular expression. For more information, see [How to combine LINQ queries with regular expressions \(C#\)](#).

Example

```

class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}

```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and File Directories \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

How to query an ArrayList with LINQ (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

When using LINQ to query non-generic `IEnumerable` collections such as `ArrayList`, you must explicitly declare the type of the range variable to reflect the specific type of the objects in the collection. For example, if you have an `ArrayList` of `Student` objects, your `from clause` should look like this:

```
var query = from Student s in arrList  
//...
```

By specifying the type of the range variable, you are casting each item in the `ArrayList` to a `Student`.

The use of an explicitly typed range variable in a query expression is equivalent to calling the `Cast` method. `Cast` throws an exception if the specified cast cannot be performed. `Cast` and `OfType` are the two Standard Query Operator methods that operate on non-generic `IEnumerable` types. For more information, see [Type Relationships in LINQ Query Operations](#).

Example

The following example shows a simple query over an `ArrayList`. Note that this example uses object initializers when the code calls the `Add` method, but this is not a requirement.

```

using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                {
                    FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                });

            var query = from Student student in arrList
                       where student.Scores[0] > 95
                       select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
   Omelchenko: 98
   Garcia: 97
*/

```

See also

- [LINQ to Objects \(C#\)](#)

How to add custom methods for LINQ queries (C#)

11/2/2020 • 5 minutes to read • [Edit Online](#)

You extend the set of methods that you use for LINQ queries by adding extension methods to the `IEnumerable<T>` interface. For example, in addition to the standard average or maximum operations, you create a custom aggregate method to compute a single value from a sequence of values. You also create a method that works as a custom filter or a specific data transform for a sequence of values and returns a new sequence. Examples of such methods are `Distinct`, `Skip`, and `Reverse`.

When you extend the `IEnumerable<T>` interface, you can apply your custom methods to any enumerable collection. For more information, see [Extension Methods](#).

Adding an aggregate method

An aggregate method computes a single value from a set of values. LINQ provides several aggregate methods, including `Average`, `Min`, and `Max`. You can create your own aggregate method by adding an extension method to the `IEnumerable<T>` interface.

The following code example shows how to create an extension method called `Median` to compute a median for a sequence of numbers of type `double`.

```
public static class LINQExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (!(source?.Any() ?? false))
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList = (from number in source
                         orderby number
                         select number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

You call this extension method for any enumerable collection in the same way you call other aggregate methods from the `IEnumerable<T>` interface.

The following code example shows how to use the `Median` method for an array of type `double`.

```

double[] numbers = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query = numbers.Median();

Console.WriteLine("double: Median = " + query);
/*
This code produces the following output:

Double: Median = 4.85
*/

```

Overloading an Aggregate Method to Accept Various Types

You can overload your aggregate method so that it accepts sequences of various types. The standard approach is to create an overload for each type. Another approach is to create an overload that will take a generic type and convert it to a specific type by using a delegate. You can also combine both approaches.

To create an overload for each type

You can create a specific overload for each type that you want to support. The following code example shows an overload of the `Median` method for the `int` type.

```

//int overload
public static double Median(this IEnumerable<int> source) =>
    (from num in source select (double)num).Median();

```

You can now call the `Median` overloads for both `integer` and `double` types, as shown in the following code:

```

double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query1 = numbers1.Median();

Console.WriteLine("double: Median = " + query1);

int[] numbers2 = { 1, 2, 3, 4, 5 };

var query2 = numbers2.Median();

Console.WriteLine("int: Median = " + query2);
/*
This code produces the following output:

Double: Median = 4.85
Integer: Median = 3
*/

```

To create a generic overload

You can also create an overload that accepts a sequence of generic objects. This overload takes a delegate as a parameter and uses it to convert a sequence of objects of a generic type to a specific type.

The following code shows an overload of the `Median` method that takes the `Func<T, TResult>` delegate as a parameter. This delegate takes an object of generic type `T` and returns an object of type `double`.

```

// Generic overload.
public static double Median<T>(this IEnumerable<T> numbers,
    Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();

```

You can now call the `Median` method for a sequence of objects of any type. If the type doesn't have its own method overload, you have to pass a delegate parameter. In C#, you can use a lambda expression for this

purpose. Also, in Visual Basic only, if you use the `Aggregate` or `Group By` clause instead of the method call, you can pass any value or expression that is in the scope this clause.

The following example code shows how to call the `Median` method for an array of integers and an array of strings. For strings, the median for the lengths of strings in the array is calculated. The example shows how to pass the `Func<T,TResult>` delegate parameter to the `Median` method for each case.

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
 You can use the num=>num lambda expression as a parameter for the Median method
 so that the compiler will implicitly convert its value to double.
 If there is no implicit conversion, the compiler will display an error message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine("int: Median = " + query3);

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of objects.

var query4 = numbers4.Median(str => str.Length);

Console.WriteLine("String: Median = " + query4);

/*
 This code produces the following output:

 Integer: Median = 3
 String: Median = 4
*/
```

Adding a method that returns a sequence

You can extend the `IEnumerable<T>` interface with a custom query method that returns a sequence of values. In this case, the method must return a collection of type `IEnumerable<T>`. Such methods can be used to apply filters or data transforms to a sequence of values.

The following example shows how to create an extension method named `AlternateElements` that returns every other element in a collection, starting from the first element.

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int i = 0;
    foreach (var element in source)
    {
        if (i % 2 == 0)
        {
            yield return element;
        }
        i++;
    }
}
```

You can call this extension method for any enumerable collection just as you would call other methods from the `IEnumerable<T>` interface, as shown in the following code:

```
string[] strings = { "a", "b", "c", "d", "e" };

var query5 = stringsAlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}

/*
This code produces the following output:

a
c
e
*/
```

See also

- [IEnumerable<T>](#)
- [Extension Methods](#)

LINQ to ADO.NET (Portal Page)

11/2/2020 • 2 minutes to read • [Edit Online](#)

LINQ to ADO.NET enables you to query over any enumerable object in ADO.NET by using the Language-Integrated Query (LINQ) programming model.

NOTE

The LINQ to ADO.NET documentation is located in the ADO.NET section of the .NET Framework SDK: [LINQ and ADO.NET](#).

There are three separate ADO.NET Language-Integrated Query (LINQ) technologies: LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet provides richer, optimized querying over the [DataSet](#), LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

LINQ to DataSet

The [DataSet](#) is one of the most widely used components in ADO.NET, and is a key element of the disconnected programming model that ADO.NET is built on. Despite this prominence, however, the [DataSet](#) has limited query capabilities.

LINQ to DataSet enables you to build richer query capabilities into [DataSet](#) by using the same query functionality that is available for many other data sources.

For more information, see [LINQ to DataSet](#).

LINQ to SQL

LINQ to SQL provides a run-time infrastructure for managing relational data as objects. In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer. When you execute the application, LINQ to SQL translates language-integrated queries in the object model into SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back into objects that you can manipulate.

LINQ to SQL includes support for stored procedures and user-defined functions in the database, and for inheritance in the object model.

For more information, see [LINQ to SQL](#).

LINQ to Entities

Through the Entity Data Model, relational data is exposed as objects in the .NET environment. This makes the object layer an ideal target for LINQ support, allowing developers to formulate queries against the database from the language used to build the business logic. This capability is known as LINQ to Entities. See [LINQ to Entities](#) for more information.

See also

- [LINQ and ADO.NET](#)
- [Language-Integrated Query \(LINQ\) \(C#\)](#)

Enabling a Data Source for LINQ Querying

3/6/2021 • 3 minutes to read • [Edit Online](#)

There are various ways to extend LINQ to enable any data source to be queried in the LINQ pattern. The data source might be a data structure, a Web service, a file system, or a database, to name some. The LINQ pattern makes it easy for clients to query a data source for which LINQ querying is enabled, because the syntax and pattern of the query does not change. The ways in which LINQ can be extended to these data sources include the following:

- Implementing the `IEnumerable<T>` interface in a type to enable LINQ to Objects querying of that type.
- Creating standard query operator methods such as `Where` and `Select` that extend a type, to enable custom LINQ querying of that type.
- Creating a provider for your data source that implements the `IQueryable<T>` interface. A provider that implements this interface receives LINQ queries in the form of expression trees, which it can execute in a custom way, for example remotely.
- Creating a provider for your data source that takes advantage of an existing LINQ technology. Such a provider would enable not only querying, but also insert, update, and delete operations and mapping for user-defined types.

This topic discusses these options.

How to Enable LINQ Querying of Your Data Source

In-Memory Data

There are two ways you can enable LINQ querying of in-memory data. If the data is of a type that implements `IEnumerable<T>`, you can query the data by using LINQ to Objects. If it does not make sense to enable enumeration of your type by implementing the `IEnumerable<T>` interface, you can define LINQ standard query operator methods in that type or create LINQ standard query operator methods that extend the type. Custom implementations of the standard query operators should use deferred execution to return the results.

Remote Data

The best option for enabling LINQ querying of a remote data source is to implement the `IQueryable<T>` interface. However, this differs from extending a provider such as LINQ to SQL for a data source.

IQueryable LINQ Providers

LINQ providers that implement `IQueryable<T>` can vary widely in their complexity. This section discusses the different levels of complexity.

A less complex `IQueryable` provider might interface with a single method of a Web service. This type of provider is very specific because it expects specific information in the queries that it handles. It has a closed type system, perhaps exposing a single result type. Most of the execution of the query occurs locally, for example by using the `Enumerable` implementations of the standard query operators. A less complex provider might examine only one method call expression in the expression tree that represents the query, and let the remaining logic of the query be handled elsewhere.

An `IQueryable` provider of medium complexity might target a data source that has a partially expressive query language. If it targets a Web service, it might interface with more than one method of the Web service and select the method to call based on the question that the query poses. A provider of medium complexity would have a

richer type system than a simple provider, but it would still be a fixed type system. For example, the provider might expose types that have one-to-many relationships that can be traversed, but it would not provide mapping technology for user-defined types.

A complex `IQueryable` provider, such as the LINQ to SQL provider, might translate complete LINQ queries to an expressive query language, such as SQL. A complex provider is more general than a less complex provider, because it can handle a wider variety of questions in the query. It also has an open type system and therefore must contain extensive infrastructure to map user-defined types. Developing a complex provider requires a significant amount of effort.

See also

- [IQueryable<T>](#)
- [IEnumerable<T>](#)
- [Enumerable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

Visual Studio IDE and Tools Support for LINQ (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Visual Studio integrated development environment (IDE) provides the following features that support LINQ application development:

Object Relational Designer

The Object Relational Designer is a visual design tool that you can use in [LINQ to SQL](#) applications to generate classes in C# that represent the relational data in an underlying database. For more information, see [LINQ to SQL Tools in Visual Studio](#).

SQLMetal Command Line Tool

SQLMetal is a command-line tool that can be used in build processes to generate classes from existing databases for use in LINQ to SQL applications. For more information, see [SqlMetal.exe \(Code Generation Tool\)](#).

LINQ-Aware Code Editor

The C# code editor supports LINQ extensively with IntelliSense and formatting capabilities.

Visual Studio Debugger Support

The Visual Studio debugger supports debugging of query expressions. For more information, see [Debugging LINQ](#).

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)

Reflection (C#)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Reflection provides objects (of type [Type](#)) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the [GetType\(\)](#) method - inherited by all types from the [Object](#) base class - to obtain the type of a variable:

NOTE

Make sure you add `using System;` and `using System.Reflection;` at the top of your .cs file.

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

The output is: `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

NOTE

The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the [IsAssembly](#) property. To identify a `protected internal` method, use the [IsFamilyOrAssembly](#).

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the topic [Dynamically Loading and Using Types](#).

Related sections

For more information:

- [Reflection](#)
- [Viewing Type Information](#)
- [Reflection and Generic Types](#)
- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

See also

- [C# Programming Guide](#)
- [Assemblies in .NET](#)

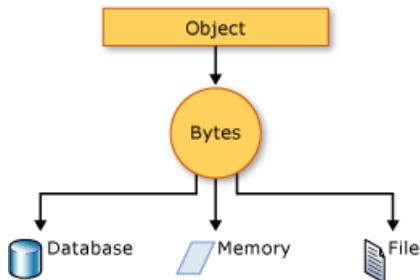
Serialization (C#)

11/2/2020 • 4 minutes to read • [Edit Online](#)

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.

Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

- Sending the object to a remote application by using a web service
- Passing an object from one domain to another
- Passing an object through a firewall as a JSON or XML string
- Maintaining security or user-specific information across applications

JSON serialization

The [System.Text.Json](#) namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.

JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to [the RFC 8259 JSON specification](#). To control the way [JsonSerializer](#) serializes or deserializes an instance of the class:

- Use a [JsonSerializerOptions](#) object
- Apply attributes from the [System.Text.Json.Serialization](#) namespace to classes or properties
- [Implement custom converters](#)

Binary and XML serialization

The [System.Runtime.Serialization](#) namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-

based network streams. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced.

WARNING

Binary serialization can be dangerous. For more information, see [BinaryFormatter security guide](#).

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. [System.Xml.Serialization](#) contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the [XmlSerializer](#) serializes or deserializes an instance of the class.

Making an object serializable

For binary or XML serialization, you need:

- The object to be serialized
- A stream to contain the serialized object
- A [System.Runtime.Serialization.Formatter](#) instance

Apply the [SerializableAttribute](#) attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the [SerializableAttribute](#) attribute.

To prevent a field from being serialized, apply the [NonSerializedAttribute](#) attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked [SerializableAttribute](#), those objects will also be serialized.

Basic and custom serialization

Binary and XML serialization can be performed in two ways, basic and custom.

Basic serialization uses .NET to automatically serialize the object. The only requirement is that the class has the [SerializableAttribute](#) attribute applied. The [NonSerializedAttribute](#) can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](#) and implement the [ISerializable](#) interface. If you want your object to be deserialized in a custom manner as well, use a custom constructor.

Designer serialization

Designer serialization is a special form of serialization that involves the kind of object persistence associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

Related Topics and Examples

[System.Text.Json overview](#) Shows how to get the `System.Text.Json` library.

[How to serialize and deserialize JSON in .NET](#) Shows how to read and write object data to and from JSON using the `JsonSerializer` class.

[Walkthrough: Persisting an Object in Visual Studio \(C#\)](#)

Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

[How to read object data from an XML file \(C#\)](#)

Shows how to read object data that was previously written to an XML file using the `XmlSerializer` class.

[How to write object data to an XML file \(C#\)](#)

Shows how to write the object from a class to an XML file using the `XmlSerializer` class.

How to write object data to an XML file (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example writes the object from a class to an XML file using the [XmlSerializer](#) class.

Example

```
public class XMLWrite
{
    static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
        "//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Compiling the Code

The class being serialized must have a public constructor without parameters.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The file exists and is read-only ([IOException](#)).
- The path is too long ([PathTooLongException](#)).
- The disk is full ([IOException](#)).

.NET Security

This example creates a new file, if the file does not already exist. If an application needs to create a file, that application needs `Create` access for the folder. If the file already exists, the application needs only `Write` access,

a lesser privilege. Where possible, it is more secure to create the file during deployment, and only grant `Read` access to a single file, rather than `Create` access for a folder.

See also

- [StreamWriter](#)
- [How to read object data from an XML file \(C#\)](#)
- [Serialization \(C#\)](#)

How to read object data from an XML file (C#)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example reads object data that was previously written to an XML file using the [XmlSerializer](#) class.

Example

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview = (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);
}
```

Compiling the Code

Replace the file name "c:\temp\SerializationOverview.xml" with the name of the file containing the serialized data. For more information about serializing data, see [How to write object data to an XML file \(C#\)](#).

The class must have a public constructor without parameters.

Only public properties and fields are deserialized.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The data in the file does not represent data from the class to be deserialized.
- The file does not exist ([IOException](#)).

.NET Security

Always verify inputs, and never deserialize data from an untrusted source. The re-created object runs on a local computer with the permissions of the code that deserialized it. Verify all inputs before using the data in your

application.

See also

- [StreamWriter](#)
- [How to write object data to an XML file \(C#\)](#)
- [Serialization \(C#\)](#)
- [C# Programming Guide](#)

Walkthrough: persisting an object using C#

3/23/2021 • 4 minutes to read • [Edit Online](#)

You can use serialization to persist an object's data between instances, which enables you to store values and retrieve them the next time that the object is instantiated.

In this walkthrough, you will create a basic `Loan` object and persist its data to a file. You will then retrieve the data from the file when you re-create the object.

IMPORTANT

This example creates a new file if the file does not already exist. If an application must create a file, that application must have `Create` permission for the folder. Permissions are set by using access control lists. If the file already exists, the application needs only `Write` permission, a lesser permission. Where possible, it's more secure to create the file during deployment and only grant `Read` permissions to a single file (instead of Create permissions for a folder). Also, it's more secure to write data to user folders than to the root folder or the Program Files folder.

IMPORTANT

This example stores data in a binary format file. These formats should not be used for sensitive data, such as passwords or credit-card information.

Prerequisites

- To build and run, install the [.NET Core SDK](#).
- Install your favorite code editor, if you haven't already.

TIP

Need to install a code editor? Try [Visual Studio](#)!

- The example requires C# 7.3. See [Select the C# language version](#)

You can examine the sample code online [at the .NET samples GitHub repository](#).

Creating the loan object

The first step is to create a `Loan` class and a console application that uses the class:

1. Create a new application. Type `dotnet new console -o serialization` to create a new console application in a subdirectory named `serialization`.
2. Open the application in your editor, and add a new class named `Loan.cs`.
3. Add the following code to your `Loan` class:

```

public class Loan : INotifyPropertyChanged
{
    public double LoanAmount { get; set; }
    public double InterestRatePercent { get; set; }

    [field:NonSerialized()]
    public DateTime TimeLastLoaded { get; set; }

    public int Term { get; set; }

    private string customer;
    public string Customer
    {
        get { return customer; }
        set
        {
            customer = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(Customer)));
        }
    }

    [field: NonSerialized()]
    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

    public Loan(double loanAmount,
               double interestRate,
               int term,
               string customer)
    {
        this.LoanAmount = loanAmount;
        this.InterestRatePercent = interestRate;
        this.Term = term;
        this.customer = customer;
    }
}

```

You will also have to create an application that uses the `Loan` class.

Serialize the loan object

1. Open `Program.cs`. Add the following code:

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

Add an event handler for the `PropertyChanged` event, and a few lines to modify the `Loan` object and display the changes. You can see the additions in the following code:

```

TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);

```

At this point, you can run the code, and see the current output:

```
New customer value: Henry Clay  
7.5  
7.1
```

Running this application repeatedly always writes the same values. A new Loan object is created every time you run the program. In the real world, interest rates change periodically, but not necessarily every time that the application is run. Serialization code means you preserve the most recent interest rate between instances of the application. In the next step, you will do just that by adding serialization to the Loan class.

Using Serialization to Persist the Object

In order to persist the values for the Loan class, you must first mark the class with the `Serializable` attribute. Add the following code above the Loan class definition:

```
[Serializable()]
```

The `SerializableAttribute` tells the compiler that everything in the class can be persisted to a file. Because the `PropertyChanged` event does not represent part of the object graph that should be stored, it should not be serialized. Doing so would serialize all objects that are attached to that event. You can add the `NonSerializedAttribute` to the field declaration for the `PropertyChanged` event handler.

```
[field: NonSerialized()]  
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

Beginning with C# 7.3, you can attach attributes to the backing field of an auto-implemented property using the `field` target value. The following code adds a `TimeLastLoaded` property and marks it as not serializable:

```
[field:NonSerialized()]  
public DateTime TimeLastLoaded { get; set; }
```

The next step is to add the serialization code to the LoanApp application. In order to serialize the class and write it to a file, you use the `System.IO` and `System.Runtime.Serialization.Formatters.Binary` namespaces. To avoid typing the fully qualified names, you can add references to the necessary namespaces as shown in the following code:

```
using System.IO;  
using System.Runtime.Serialization.Formatters.Binary;
```

The next step is to add code to deserialize the object from the file when the object is created. Add a constant to the class for the serialized data's file name as shown in the following code:

```
const string FileName = @"../../SavedLoan.bin";
```

Next, add the following code after the line that creates the `TestLoan` object:

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

You first must check that the file exists. If it exists, create a [Stream](#) class to read the binary file and a [BinaryFormatter](#) class to translate the file. You also need to convert from the stream type to the Loan object type.

Next you must add code to serialize the class to a file. Add the following code after the existing code in the [Main](#) method:

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

At this point, you can again build and run the application. The first time it runs, notice that the interest rates starts at 7.5, and then changes to 7.1. Close the application and then run it again. Now, the application prints the message that it has read the saved file, and the interest rate is 7.1 even before the code that changes it.

See also

- [Serialization \(C#\)](#)
- [C# Programming Guide](#)

Statements, Expressions, and Operators (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The C# code that comprises an application consists of statements made up of keywords, expressions and operators. This section contains information regarding these fundamental elements of a C# program.

For more information, see:

- [Statements](#)
- [Operators and expressions](#)
- [Expression-bodied members](#)
- [Anonymous Functions](#)
- [Equality Comparisons](#)

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Casting and Type Conversions](#)

Statements (C# Programming Guide)

11/2/2020 • 6 minutes to read • [Edit Online](#)

The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

```
static void Main()
{
    // Declaration statement.
    int counter;

    // Assignment statement.
    counter = 1;

    // Error! This is an expression, not an expression statement.
    // counter + 1;

    // Declaration statements with initializers are functionally
    // equivalent to declaration statement followed by assignment statement:
    int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
    const double pi = 3.14159; // Declare and initialize constant.

    // foreach statement block that contains multiple statements.
    foreach (int radius in radii)
    {
        // Declaration statement with initializer.
        double circumference = pi * (2 * radius);

        // Expression statement (method invocation). A single-line
        // statement can span multiple text lines because line breaks
        // are treated as white space, which is ignored by the compiler.
        System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                               counter, radius, circumference);

        // Expression statement (postfix increment).
        counter++;
    } // End of foreach statement block
} // End of Main method body.
} // End of SimpleStatements class.
/*
Output:
Radius of circle #1 = 15. Circumference = 94.25
Radius of circle #2 = 32. Circumference = 201.06
Radius of circle #3 = 108. Circumference = 678.58
Radius of circle #4 = 74. Circumference = 464.96
Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types of statements

The following table lists the various types of statements in C# and their associated keywords, with links to topics

that include more information:

CATEGORY	C# KEYWORDS / NOTES
Declaration statements	A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.
Expression statements	Expression statements that calculate a value must store the value in a variable.
Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics: <ul style="list-style-type: none">• if• else• switch• case
Iteration statements	Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics: <ul style="list-style-type: none">• do• for• foreach• in• while
Jump statements	Jump statements transfer control to another section of code. For more information, see the following topics: <ul style="list-style-type: none">• break• continue• default• goto• return• yield
Exception handling statements	Exception handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics: <ul style="list-style-type: none">• throw• try-catch• try-finally• try-catch-finally
Checked and unchecked	Checked and unchecked statements enable you to specify whether numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value. For more information, see checked and unchecked .

CATEGORY	C# KEYWORDS / NOTES
The <code>await</code> statement	<p>If you mark a method with the <code>async</code> modifier, you can use the <code>await</code> operator in the method. When control reaches an <code>await</code> expression in the <code>async</code> method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.</p> <p>For a simple example, see the "Async Methods" section of Methods. For more information, see Asynchronous Programming with <code>async</code> and <code>await</code>.</p>
The <code>yield return</code> statement	<p>An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the <code>yield return</code> statement to return each element one at a time. When a <code>yield return</code> statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.</p> <p>For more information, see Iterators.</p>
The <code>fixed</code> statement	<p>The <code>fixed</code> statement prevents the garbage collector from relocating a movable variable. For more information, see fixed.</p>
The <code>lock</code> statement	<p>The <code>lock</code> statement enables you to limit access to blocks of code to only one thread at a time. For more information, see lock.</p>
Labeled statements	<p>You can give a statement a label and then use the <code>goto</code> keyword to jump to the labeled statement. (See the example in the following row.)</p>
The <code>empty</code> statement	<p>The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed.</p>

Declaration statements

The following code shows examples of variable declarations with and without an initial assignment, and a constant declaration with the necessary initialization.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Expression statements

The following code shows examples of expression statements, including assignment, object creation with assignment, and method invocation.

```

// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();

```

The empty statement

The following examples show two uses for an empty statement:

```

void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}

```

Embedded statements

Some statements, including [do](#), [while](#), [for](#), and [foreach](#), always have an embedded statement that follows them. This embedded statement may be either a single statement or multiple statements enclosed by {} brackets in a statement block. Even single-line embedded statements can be enclosed in {} brackets, as shown in the following example:

```

// Recommended style. Embedded statement in block.
foreach (string s in System.IO.Directory.GetDirectories(
            System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
            System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);

```

An embedded statement that is not enclosed in {} brackets cannot be a declaration statement or a labeled statement. This is shown in the following example:

```

if(pointB == true)
    //Error CS1023:
    int radius = 5;

```

Put the embedded statement in a block to fix the error:

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToString());
}
```

Nested statement blocks

Statement blocks can be nested, as shown in the following code:

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.;"
```

Unreachable statements

If the compiler determines that the flow of control can never reach a particular statement under any circumstances, it will produce warning CS0162, as shown in the following example:

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

C# language specification

For more information, see the [Statements](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Statement keywords](#)
- [C# operators and expressions](#)

Expression-bodied members (C# programming guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Expression body definitions let you provide a member's implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression. An expression body definition has the following general syntax:

```
member => expression;
```

where *expression* is a valid expression.

Support for expression body definitions was introduced for methods and read-only properties in C# 6 and was expanded in C# 7.0. Expression body definitions can be used with the type members listed in the following table:

MEMBER	SUPPORTED AS OF...
Method	C# 6
Read-only property	C# 6
Property	C# 7.0
Constructor	C# 7.0
Finalizer	C# 7.0
Indexer	C# 7.0

Methods

An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for methods that return `void`, that performs some operation. For example, types that override the `ToString` method typically include a single expression that returns the string representation of the current object.

The following example defines a `Person` class that overrides the `ToString` method with an expression body definition. It also defines a `DisplayName` method that displays a name to the console. Note that the `return` keyword is not used in the `ToString` expression body definition.

```

using System;

public class Person
{
    public Person(string firstName, string lastName)
    {
        fname = firstName;
        lname = lastName;
    }

    private string fname;
    private string lname;

    public override string ToString() => $"{fname} {lname}".Trim();
    public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
    static void Main()
    {
        Person p = new Person("Mandy", "Dejesus");
        Console.WriteLine(p);
        p.DisplayName();
    }
}

```

For more information, see [Methods \(C# Programming Guide\)](#).

Read-only properties

Starting with C# 6, you can use expression body definition to implement a read-only property. To do that, use the following syntax:

```
.PropertyType PropertyName => expression;
```

The following example defines a `Location` class whose read-only `Name` property is implemented as an expression body definition that returns the value of the private `locationName` field:

```

public class Location
{
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}

```

For more information about properties, see [Properties \(C# Programming Guide\)](#).

Properties

Starting with C# 7.0, you can use expression body definitions to implement property `get` and `set` accessors. The following example demonstrates how to do that:

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

For more information about properties, see [Properties \(C# Programming Guide\)](#).

Constructors

An expression body definition for a constructor typically consists of a single assignment expression or a method call that handles the constructor's arguments or initializes instance state.

The following example defines a `Location` class whose constructor has a single string parameter named `name`. The expression body definition assigns the argument to the `Name` property.

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

For more information, see [Constructors \(C# Programming Guide\)](#).

Finalizers

An expression body definition for a finalizer typically contains cleanup statements, such as statements that release unmanaged resources.

The following example defines a finalizer that uses an expression body definition to indicate that the finalizer has been called.

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

For more information, see [Finalizers \(C# Programming Guide\)](#).

Indexers

Like with properties, indexer `get` and `set` accessors consist of expression body definitions if the `get` accessor consists of a single expression that returns a value or the `set` accessor performs a simple assignment.

The following example defines a class named `Sports` that includes an internal `String` array that contains the names of a number of sports. Both the indexer `get` and `set` accessors are implemented as expression body definitions.

```
using System;
using System.Collections.Generic;

public class Sports
{
    private string[] types = { "Baseball", "Basketball", "Football",
                               "Hockey", "Soccer", "Tennis",
                               "Volleyball" };

    public string this[int i]
    {
        get => types[i];
        set => types[i] = value;
    }
}
```

For more information, see [Indexers \(C# Programming Guide\)](#).

Anonymous functions (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An anonymous function is an "inline" statement or expression that can be used wherever a delegate type is expected. You can use it to initialize a named delegate or pass it instead of a named delegate type as a method parameter.

You can use a [lambda expression](#) or an [anonymous method](#) to create an anonymous function. We recommend using lambda expressions as they provide more concise and expressive way to write inline code. Unlike anonymous methods, some types of lambda expressions can be converted to the expression tree types.

The Evolution of Delegates in C#

In C# 1.0, you created an instance of a delegate by explicitly initializing it with a method that was defined elsewhere in the code. C# 2.0 introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 introduced lambda expressions, which are similar in concept to anonymous methods but more expressive and concise. These two features are known collectively as *anonymous functions*. In general, applications that target .NET Framework 3.5 or later should use lambda expressions.

The following example demonstrates the evolution of delegate creation from C# 1.0 to C# 3.0:

```

class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Hello. My name is M and I write lines.
That's nothing. I'm anonymous and
I'm a famous author.
Press any key to exit.
*/

```

C# language specification

For more information, see the [Anonymous function expressions](#) section of the [C# language specification](#).

See also

- [Statements, Expressions, and Operators](#)
- [Lambda Expressions](#)
- [Delegates](#)
- [Expression Trees \(C#\)](#)

How to use lambda expressions in a query (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You do not use lambda expressions directly in query syntax, but you do use them in method calls, and query expressions can contain method calls. In fact, some query operations can only be expressed in method syntax. For more information about the difference between query syntax and method syntax, see [Query Syntax and Method Syntax in LINQ](#).

Example

The following example demonstrates how to use a lambda expression in a method-based query by using the [Enumerable.Where](#) standard query operator. Note that the [Where](#) method in this example has an input parameter of the delegate type [Func<T,TResult>](#) and that delegate takes an integer as input and returns a Boolean. The lambda expression can be converted to that delegate. If this were a LINQ to SQL query that used the [Queryable.Where](#) method, the parameter type would be an [Expression<Func<int,bool>>](#) but the lambda expression would look exactly the same. For more information on the Expression type, see [System.Linq.Expressions.Expression](#).

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

Example

The following example demonstrates how to use a lambda expression in a method call of a query expression. The lambda is necessary because the [Sum](#) standard query operator cannot be invoked by using query syntax.

The query first groups the students according to their grade level, as defined in the [GradeLevel](#) enum. Then for each group it adds the total scores for each student. This requires two [Sum](#) operations. The inner [Sum](#) calculates the total score for each student, and the outer [Sum](#) keeps a running, combined total for all students in the group.

```

private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
        from student in students
        group student by student.Year into studentGroup
        select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
Outputs:
Key = SecondYear Sum = 1014
Key = ThirdYear Sum = 964
Key = FirstYear Sum = 1058
Key = FourthYear Sum = 974
*/

```

Compiling the Code

To run this code, copy and paste the method into the `StudentClass` that is provided in [Query a collection of objects](#) and call it from the `Main` method.

See also

- [Lambda Expressions](#)
- [Expression Trees \(C#\)](#)

Equality comparisons (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

It is sometimes necessary to compare two values for equality. In some cases, you are testing for *value equality*, also known as *equivalence*, which means that the values that are contained by the two variables are equal. In other cases, you have to determine whether two variables refer to the same underlying object in memory. This type of equality is called *reference equality*, or *identity*. This topic describes these two kinds of equality and provides links to other topics for more information.

Reference equality

Reference equality means that two object references refer to the same underlying object. This can occur through simple assignment, as shown in the following example.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

In this code, two objects are created, but after the assignment statement, both references refer to the same object. Therefore they have reference equality. Use the [ReferenceEquals](#) method to determine whether two references refer to the same object.

The concept of reference equality applies only to reference types. Value type objects cannot have reference equality because when an instance of a value type is assigned to a variable, a copy of the value is made. Therefore you can never have two unboxed structs that refer to the same location in memory. Furthermore, if you use [ReferenceEquals](#) to compare two value types, the result will always be `false`, even if the values that are contained in the objects are all identical. This is because each variable is boxed into a separate object instance. For more information, see [How to test for reference equality \(Identity\)](#).

Value equality

Value equality means that two objects contain the same value or values. For primitive value types such as `int` or `bool`, tests for value equality are straightforward. You can use the `==` operator, as shown in the following example.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

For most other types, testing for value equality is more complex because it requires that you understand how the type defines it. For classes and structs that have multiple fields or properties, value equality is often defined to mean that all fields or properties have the same value. For example, two `Point` objects might be defined to be equivalent if `pointA.X` is equal to `pointB.X` and `pointA.Y` is equal to `pointB.Y`. For records, value equality means that two variables of a record type are equal if the types match and all property and field values match.

However, there is no requirement that equivalence be based on all the fields in a type. It can be based on a subset. When you compare types that you do not own, you should make sure to understand specifically how equivalence is defined for that type. For more information about how to define value equality in your own classes and structs, see [How to define value equality for a type](#).

Value equality for floating-point values

Equality comparisons of floating-point values (`double` and `float`) are problematic because of the imprecision of floating-point arithmetic on binary computers. For more information, see the remarks in the topic [System.Double](#).

Related topics

TITLE	DESCRIPTION
How to test for reference equality (Identity)	Describes how to determine whether two variables have reference equality.
How to define value equality for a type	Describes how to provide a custom definition of value equality for a type.
C# Programming Guide	Provides links to detailed information about important C# language features and features that are available to C# through .NET.
Types	Provides information about the C# type system and links to additional information.
Records	Provides information about record types, which test for value equality by default.

See also

- [C# Programming Guide](#)

How to define value equality for a class or struct (C# Programming Guide)

3/31/2021 • 9 minutes to read • [Edit Online](#)

Records automatically implement value equality. Consider defining a `record` instead of a `class` when your type models data and should implement value equality.

When you define a class or struct, you decide whether it makes sense to create a custom definition of value equality (or equivalence) for the type. Typically, you implement value equality when you expect to add objects of the type to a collection, or when their primary purpose is to store a set of fields or properties. You can base your definition of value equality on a comparison of all the fields and properties in the type, or you can base the definition on a subset.

In either case, and in both classes and structs, your implementation should follow the five guarantees of equivalence (for the following rules, assume that `x`, `y` and `z` are not null):

1. The reflexive property: `x.Equals(x)` returns `true`.
2. The symmetric property: `x.Equals(y)` returns the same value as `y.Equals(x)`.
3. The transitive property: if `(x.Equals(y) && y.Equals(z))` returns `true`, then `x.Equals(z)` returns `true`.
4. Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` aren't modified.
5. Any non-null value isn't equal to null. However, `x.Equals(y)` throws an exception when `x` is null. That breaks rules 1 or 2, depending on the argument to `Equals`.

Any struct that you define already has a default implementation of value equality that it inherits from the `System.ValueType` override of the `Object.Equals(Object)` method. This implementation uses reflection to examine all the fields and properties in the type. Although this implementation produces correct results, it is relatively slow compared to a custom implementation that you write specifically for the type.

The implementation details for value equality are different for classes and structs. However, both classes and structs require the same basic steps for implementing equality:

1. Override the `virtual Object.Equals(Object)` method. In most cases, your implementation of

```
bool Equals( object obj )
```

 should just call into the type-specific `Equals` method that is the implementation of the `System.IEquatable<T>` interface. (See step 2.)
2. Implement the `System.IEquatable<T>` interface by providing a type-specific `Equals` method. This is where the actual equivalence comparison is performed. For example, you might decide to define equality by comparing only one or two fields in your type. Don't throw exceptions from `Equals`. For classes that are related by inheritance:
 - This method should examine only fields that are declared in the class. It should call `base.Equals` to examine fields that are in the base class. (Don't call `base.Equals` if the type inherits directly from `Object`, because the `Object` implementation of `Object.Equals(Object)` performs a reference equality check.)
 - Two variables should be deemed equal only if the run-time types of the variables being compared are the same. Also, make sure that the `IEquatable` implementation of the `Equals` method for the run-time type is used if the run-time and compile-time types of a variable are different. One

strategy for making sure run-time types are always compared correctly is to implement `IEquatable` only in `sealed` classes. For more information, see the [class example](#) later in this article.

3. Optional but recommended: Overload the `==` and `!=` operators.
4. Override `Object.GetHashCode` so that two objects that have value equality produce the same hash code.
5. Optional: To support definitions for "greater than" or "less than," implement the `IComparable<T>` interface for your type, and also overload the `<=` and `>=` operators.

NOTE

Starting in C# 9.0, you can use records to get value equality semantics without any unnecessary boilerplate code.

Class example

The following example shows how to implement value equality in a class (reference type).

```
using System;

namespace ValueEqualityClass
{
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj) => this.Equals(obj as TwoDPoint);

        public bool Equals(TwoDPoint p)
        {
            if (p is null)
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }
    }
}
```

```
        ,  
  
    public override int GetHashCode() => (X, Y).GetHashCode();  
  
    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)  
{  
    if (lhs is null)  
    {  
        if (rhs is null)  
        {  
            return true;  
        }  
  
        // Only the left side is null.  
        return false;  
    }  
    // Equals handles case of null on right side.  
    return lhs.Equals(rhs);  
}  
  
    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);  
}  
  
// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.  
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>  
{  
    public int Z { get; private set; }  
  
    public ThreeDPoint(int x, int y, int z)  
        : base(x, y)  
    {  
        if ((z < 1) || (z > 2000))  
        {  
            throw new ArgumentException("Point must be in range 1 - 2000");  
        }  
        this.Z = z;  
    }  
  
    public override bool Equals(object obj) => this.Equals(obj as ThreeDPoint);  
  
    public bool Equals(ThreeDPoint p)  
    {  
        if (p is null)  
        {  
            return false;  
        }  
  
        // Optimization for a common success case.  
        if (Object.ReferenceEquals(this, p))  
        {  
            return true;  
        }  
  
        // Check properties that this class declares.  
        if (Z == p.Z)  
        {  
            // Let base class check its own fields  
            // and do the run-time type comparison.  
            return base.Equals((TwoDPoint)p);  
        }  
        else  
        {  
            return false;  
        }  
    }  
  
    public override int GetHashCode() => (X, Y, Z).GetHashCode();  
  
    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)  
    {
```

```

    if (lhs is null)
    {
        if (rhs is null)
        {
            // null == null = true.
            return true;
        }

        // Only the left side is null.
        return false;
    }
    // Equals handles the case of null on right side.
    return lhs.Equals(rhs);
}

public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   pointA.Equals(pointB) = True
   pointA == pointB = True
   null comparison = False
   Compare to some other type = False
   Two null TwoDPoints are equal: True
   (pointE == pointA) = False
   (pointA == pointE) = False
   (pointA != pointE) = True
   pointE.Equals(list[0]): False
*/
}

```

On classes (reference types), the default implementation of both [Object.Equals\(Object\)](#) methods performs a reference equality comparison, not a value equality check. When an implementer overrides the virtual method,

the purpose is to give it value equality semantics.

The `==` and `!=` operators can be used with classes even if the class does not overload them. However, the default behavior is to perform a reference equality check. In a class, if you overload the `Equals` method, you should overload the `==` and `!=` operators, but it is not required.

IMPORTANT

The preceding example code may not handle every inheritance scenario the way you expect. Consider the following code:

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

This code reports that `p1` equals `p2` despite the difference in `z` values. The difference is ignored because the compiler picks the `TwoDPoint` implementation of `IEquatable` based on the compile-time type.

The built-in value equality of `record` types handles scenarios like this correctly. If `TwoDPoint` and `ThreeDPoint` were `record` types, the result of `p1.Equals(p2)` would be `False`. For more information, see [Equality in record type inheritance hierarchies](#).

Struct example

The following example shows how to implement value equality in a struct (value type):

```
using System;

namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object obj) => obj is TwoDPoint other && this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) => lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
        }
    }
}
```

```

int i = 5;

// True:
Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
// True:
Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
// True:
Console.WriteLine("object.Equals(pointA, pointB) = {0}", object.Equals(pointA, pointB));
// False:
Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
// False:
Console.WriteLine("(pointA == null) = {0}", pointA == null);
// True:
Console.WriteLine("(pointA != null) = {0}", pointA != null);
// False:
Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
// CS0019:
// Console.WriteLine("pointA == i = {0}", pointA == i);

// Compare unboxed to boxed.
System.Collections.ArrayList list = new System.Collections.ArrayList();
list.Add(new TwoDPoint(3, 4));
// True:
Console.WriteLine("pointA.Equals(list[0]): {0}", pointA.Equals(list[0]));

// Compare nullable to nullable and to non-nullable.
TwoDPoint? pointC = null;
TwoDPoint? pointD = null;
// False:
Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
// True:
Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

TwoDPoint temp = new TwoDPoint(3, 4);
pointC = temp;
// True:
Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

pointD = temp;
// True:
Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

}

/* Output:
pointA.Equals(pointB) = True
pointA == pointB = True
Object.Equals(pointA, pointB) = True
pointA.Equals(null) = False
(pointA == null) = False
(pointA != null) = True
pointA.Equals(i) = False
pointE.Equals(list[0]): True
pointA == (pointC = null) = False
pointC == pointD = True
pointA == (pointC = 3,4) = True
pointD == (pointC = 3,4) = True
*/
}

```

For structs, the default implementation of [Object.Equals\(Object\)](#) (which is the overridden version in [System.ValueType](#)) performs a value equality check by using reflection to compare the values of every field in the type. When an implementer overrides the virtual `Equals` method in a struct, the purpose is to provide a more efficient means of performing the value equality check and optionally to base the comparison on some

subset of the struct's field or properties.

The `==` and `!=` operators can't operate on a struct unless the struct explicitly overloads them.

See also

- [Equality comparisons](#)
- [C# programming guide](#)

How to test for reference equality (Identity) (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

You do not have to implement any custom logic to support reference equality comparisons in your types. This functionality is provided for all types by the static `Object.ReferenceEquals` method.

The following example shows how to determine whether two variables have *reference equality*, which means that they refer to the same object in memory.

The example also shows why `Object.ReferenceEquals` always returns `false` for value types and why you should not use `ReferenceEquals` to determine string equality.

Example

```
using System;
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.

            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) = {0}",
                Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
        }
    }
}
```

```

        // that have reference equality also have value equality.
        tcA.Num = 42;
        tcA.Name = "TestClass 42";
        Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
        #endregion

        // Demonstrate that two value type instances never have reference equality.
        #region ValueTypes

        TestStruct tsC = new TestStruct( 1, "TestStruct 1");

        // Value types are copied on assignment. tsD and tsC have
        // the same values but are not the same object.
        TestStruct tsD = tsC;
        Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) = {0}",
                          Object.ReferenceEquals(tsC, tsD)); // false
        #endregion

        #region stringRefEquality
        // Constant strings within the same assembly are always interned by the runtime.
        // This means they are stored in the same location in memory. Therefore,
        // the two strings have reference equality although no assignment takes place.
        string strA = "Hello world!";
        string strB = "Hello world!";
        Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                          Object.ReferenceEquals(strA, strB)); // true

        // After a new string is assigned to strA, strA and strB
        // are no longer interned and no longer have reference equality.
        strA = "Goodbye world!";
        Console.WriteLine("strA = \'{0}\\' strB = \'{1}\\'", strA, strB);

        Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
                          Object.ReferenceEquals(strA, strB)); // false

        // A string that is created at runtime cannot be interned.
        StringBuilder sb = new StringBuilder("Hello world!");
        string stringC = sb.ToString();
        // False:
        Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                          Object.ReferenceEquals(stringC, strB));

        // The string class overloads the == operator to perform an equality comparison.
        Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

        #endregion

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
}

/* Output:
   ReferenceEquals(tcA, tcB) = False
   After assignment: ReferenceEquals(tcA, tcB) = True
   tcB.Name = TestClass 42 tcB.Num: 42
   After assignment: ReferenceEquals(tsC, tsD) = False
   ReferenceEquals(strA, strB) = True
   strA = "Goodbye world!" strB = "Hello world!"
   After strA changes, ReferenceEquals(strA, strB) = False
   ReferenceEquals(stringC, strB) = False
   stringC == strB = True
*/

```

The implementation of `Equals` in the [System.Object](#) universal base class also performs a reference equality

check, but it is best not to use this because, if a class happens to override the method, the results might not be what you expect. The same is true for the `==` and `!=` operators. When they are operating on reference types, the default behavior of `==` and `!=` is to perform a reference equality check. However, derived classes can overload the operator to perform a value equality check. To minimize the potential for error, it is best to always use [ReferenceEquals](#) when you have to determine whether two objects have reference equality.

Constant strings within the same assembly are always interned by the runtime. That is, only one instance of each unique literal string is maintained. However, the runtime does not guarantee that strings created at runtime are interned, nor does it guarantee that two equal constant strings in different assemblies are interned.

See also

- [Equality Comparisons](#)

Types (C# Programming Guide)

3/6/2021 • 12 minutes to read • [Edit Online](#)

Types, variables, and values

C# is a strongly typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method declaration specifies a name, number of parameters, and type and kind (value, reference, or output) for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types and more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library and user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following items:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The interface(s) it implements.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure all operations that are performed in your code are *type safe*. For example, if you declare a variable of type `int`, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type `bool`, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

NOTE

C and C++ developers, notice that in C#, `bool` is not convertible to `int`.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying types in variable declarations

When you declare a variable or constant in a program, you must either specify its type or use the `var` keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:  
float temperature;  
string name;  
MyClass myClass;  
  
// Declaration with initializers (four examples):  
char firstLetter = 'C';  
var limit = 3;  
int[] source = { 0, 1, 2, 3, 4, 5 };  
var query = from item in source  
            where item <= limit  
            select item;
```

The types of method parameters and return values are specified in the method declaration. The following signature shows a method that requires an [int](#) as an input argument and returns a string:

```
public string GetName(int ID)  
{  
    if (ID < names.Length)  
        return names[ID];  
    else  
        return String.Empty;  
}  
private string[] names = { "Spencer", "Sally", "Doug" };
```

After you declare a variable, you can't redeclare it with a new type, and you can't assign a value not compatible with its declared type. For example, you can't declare an [int](#) and then assign it a Boolean value of [true](#). However, values can be converted to other types, for example when they're assigned to new variables or passed as method arguments. A *type conversion* that doesn't cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see [Casting and Type Conversions](#).

Built-in types

C# provides a standard set of built-in types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in [string](#) and [object](#) types. These types are available for you to use in any C# program. For the complete list of the built-in types, see [Built-in types](#).

Custom types

You use the [struct](#), [class](#), [interface](#), [enum](#), and [record](#) constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they're defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see [.NET Class Library](#).

The common type system

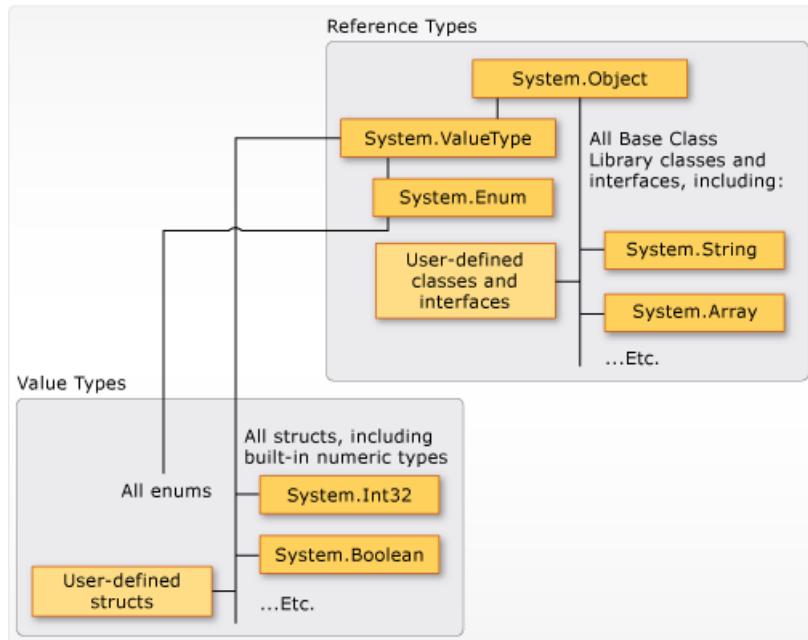
It's important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as [System.Int32](#) (C#

keyword: `int`), derive ultimately from a single base type, which is `System.Object` (C# keyword: `object`). This unified type hierarchy is called the [Common Type System](#) (CTS). For more information about inheritance in C#, see [Inheritance](#).

- Each type in the CTS is defined as either a *value type* or a *reference type*. These types include all custom types in the .NET class library and also your own user-defined types. Types that you define by using the `struct` keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the `class` or `record` keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



NOTE

You can see that the most commonly used types are all organized in the `System` namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value types

Value types derive from `System.ValueType`, which derives from `System.Object`. Types that derive from `System.ValueType` have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There's no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: `struct` and `enum`.

The built-in numeric types are structs, and they have fields and methods that you can access:

```
// constant field on type byte.  
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they're simple non-aggregate types:

```
byte num = 0xA;  
int i = 5;  
char c = 'Z';
```

Value types are *sealed*, which means that you can't derive a type from any value type, for example `System.Int32`.

You can't define a struct to inherit from any user-defined class or struct because a struct can only inherit from [System.ValueType](#). However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this cast causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a [System.Object](#) or any interface type as an input parameter. For more information, see [Boxing and Unboxing](#).

You use the [struct](#) keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

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

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see [Structure types](#). For more information about value types, see [Value types](#).

The other category of value types is [enum](#). An enum defines a set of named integral constants. For example, the [System.IO.FileMode](#) enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It's defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it's better to use enumerations instead of constant literal numbers. For more information, see [System.IO.FileMode](#).

All enums inherit from [System.Enum](#), which inherits from [System.ValueType](#). All the rules that apply to structs also apply to enums. For more information about enums, see [Enumeration types](#).

Reference types

A type that is defined as a [class](#), [record](#), [delegate](#), array, or [interface](#) is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value `null` until you explicitly create an object by using the `new` operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they're allocated and when they're reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it doesn't create a performance issue. For more information about garbage collection, see [Automatic Memory Management](#).

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the [System.Array](#) class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.  
int[] nums = { 1, 2, 3, 4, 5 };  
  
// Access an instance property of System.Array.  
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that isn't defined as [sealed](#), and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see [Classes and Structs](#). For more information about inheritance and virtual methods, see [Inheritance](#).

Types of literal values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see [Integral numeric types](#) and [Floating-point numeric types](#).

Because literals are typed, and all types derive ultimately from [System.Object](#), you can write and compile code such as the following code:

```
string s = "The answer is " + 5.ToString();  
// Outputs: "The answer is 5"  
Console.WriteLine(s);  
  
Type type = 12345.GetType();  
// Outputs: "System.Int32"  
Console.WriteLine(type);
```

Generic types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type [System.Collections.Generic.List<T>](#) has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to `object`. Generic collection classes are called *strongly typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see [Generics](#).

Implicit types, anonymous types, and nullable value types

As stated previously, you can implicitly type a local variable (but not class members) by using the `var` keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see [Implicitly Typed Local Variables](#).

It can be inconvenient to create a named type for simple sets of related values that you don't intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see [Anonymous Types](#).

Ordinary value types can't have a value of `null`. However, you can create nullable value types by appending a `?` after the type. For example, `int?` is an `int` type that can also have the value `null`. Nullable value types are instances of the generic struct type `System.Nullable<T>`. Nullable value types are especially useful when you're passing data to and from databases in which numeric values might be null. For more information, see [Nullable value types](#).

Compile-time type and runtime type

A variable can have different compile-time and run-time types. The *compile-time type* is the declared or inferred type of the variable in the source code. The *run-time type* is the type of the instance referred to by that variable. Often those two types are the same, as in the following example:

```
string message = "This is a string of characters";
```

In other cases, the compile-time type is different, as shown in the following two examples:

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

In both of the preceding examples, the run-time type is a `string`. The compile-time type is `object` in the first line, and `IEnumerable<char>` in the second.

If the two types are different for a variable, it's important to understand when the compile-time type and the run-time type apply. The compile-time type determines all the actions taken by the compiler. These compiler actions include method call resolution, overload resolution, and available implicit and explicit casts. The run-time type determines all actions that are resolved at run time. These run-time actions include dispatching virtual method calls, evaluating `is` and `switch` expressions, and other type testing APIs. To better understand how your code interacts with types, recognize which action applies to which type.

Related sections

For more information, see the following articles:

- [Casting and Type Conversions](#)
- [Boxing and Unboxing](#)
- [Using Type dynamic](#)
- [Value Types](#)
- [Reference Types](#)
- [Classes and Structs](#)
- [Anonymous Types](#)
- [Generics](#)

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Conversion of XML Data Types](#)
- [Integral types](#)

Casting and type conversions (C# Programming Guide)

11/2/2020 • 5 minutes to read • [Edit Online](#)

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or assigned a value of another type unless that type is implicitly convertible to the variable's type. For example, the `string` cannot be implicitly converted to `int`. Therefore, after you declare `i` as an `int`, you cannot assign the string "Hello" to it, as the following code shows:

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as `double`. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called *type conversions*. In C#, you can perform the following kinds of conversions:

- **Implicit conversions:** No special syntax is required because the conversion always succeeds and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts):** Explicit conversions require a [cast expression](#). Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined conversions:** User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class-derived class relationship. For more information, see [User-defined conversion operators](#).
- **Conversions with helper classes:** To convert between non-compatible types, such as integers and `System.DateTime` objects, or hexadecimal strings and byte arrays, you can use the `System.BitConverter` class, the `System.Convert` class, and the `Parse` methods of the built-in numeric types, such as `Int32.Parse`. For more information, see [How to convert a byte array to an int](#), [How to convert a string to a number](#), and [How to convert between hexadecimal strings and numeric types](#).

Implicit conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For integral types, this means the range of the source type is a proper subset of the range for the target type. For example, a variable of type `long` (64-bit integer) can store any value that an `int` (32-bit integer) can store. In the following example, the compiler implicitly converts the value of `num` on the right to a type `long` before assigning it to `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

For a complete list of all implicit numeric conversions, see the [Implicit numeric conversions](#) section of the [Built-in numeric conversions](#) article.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Explicit conversions

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur, or the cast may fail at runtime. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a [double](#) to an [int](#). The program will not compile without the cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

For a complete list of supported explicit numeric conversions, see the [Explicit numeric conversions](#) section of the [Built-in numeric conversions](#) article.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see

Type conversion exceptions at run time

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an [InvalidOperationException](#) to be thrown.

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

The `Test` method has an `Animal` parameter, thus explicitly casting the argument `a` to a `Reptile` makes a dangerous assumption. It is safer to not make assumptions, but rather check the type. C# provides the `is` operator to enable you to test for compatibility before actually performing a cast. For more information, see [How to safely cast using pattern matching and the as and is operators](#).

C# language specification

For more information, see the [Conversions](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Types](#)
- [Cast expression](#)
- [User-defined conversion operators](#)
- [Generalized Type Conversion](#)
- [How to convert a string to a number](#)

Boxing and Unboxing (C# Programming Guide)

11/2/2020 • 4 minutes to read • [Edit Online](#)

Boxing is the process of converting a [value type](#) to the type `object` or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a `System.Object` instance and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

In the following example, the integer variable `i` is *boxed* and assigned to object `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

The object `o` can then be unboxed and assigned to integer variable `i`:

```
o = 123;
i = (int)o; // unboxing
```

The following examples illustrate how boxing is used in C#.

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
646F74636574
*/
```

Performance

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see [Performance](#).

Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a [value type](#) to the type `object` or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

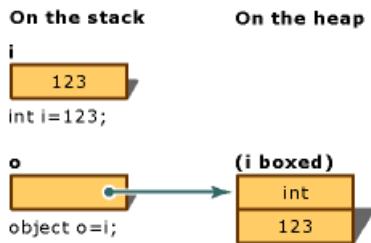
Consider the following declaration of a value-type variable:

```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable `i`:

```
// Boxing copies the value of i into object o.  
object o = i;
```

The result of this statement is creating an object reference `o`, on the stack, that references a value of the type `int`, on the heap. This value is a copy of the value-type value assigned to the variable `i`. The difference between the two variables, `i` and `o`, is illustrated in the following image of boxing conversion:



It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;  
object o = (object)i; // explicit boxing
```

Description

This example converts an integer variable `i` to an object `o` by using boxing. Then, the value stored in the variable `i` is changed from `123` to `456`. The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

Example

```
class TestBoxing  
{  
    static void Main()  
    {  
        int i = 123;  
  
        // Boxing copies the value of i into object o.  
        object o = i;  
  
        // Change the value of i.  
        i = 456;  
  
        // The change in i doesn't affect the value stored in o.  
        System.Console.WriteLine("The value-type value = {0}", i);  
        System.Console.WriteLine("The object-type value = {0}", o);  
    }  
}  
/* Output:  
   The value-type value = 456  
   The object-type value = 123  
*/
```

Unboxing

Unboxing is an explicit conversion from the type `object` to a `value type` or from an interface type to a value type that implements the interface. An unboxing operation consists of:

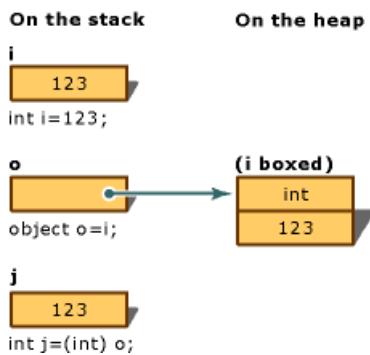
- Checking the object instance to make sure that it is a boxed value of the given value type.

- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;      // a value type
object o = i;      // boxing
int j = (int)o;    // unboxing
```

The following figure demonstrates the result of the previous statements:



For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox `null` causes a [NullReferenceException](#). Attempting to unbox a reference to an incompatible value type causes an [InvalidCastException](#).

Example

The following example demonstrates a case of invalid unboxing and the resulting `InvalidOperationException`. Using `try` and `catch`, an error message is displayed when the error occurs.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i; // implicit boxing

        try
        {
            int j = (short)o; // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

This program outputs:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

If you change the statement:

```
int j = (short) o;
```

to:

```
int j = (int) o;
```

the conversion will be performed, and you will get the output:

```
Unboxing OK.
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# programming guide](#)
- [Reference types](#)
- [Value types](#)

How to convert a byte array to an int (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

This example shows you how to use the [BitConverter](#) class to convert an array of bytes to an [int](#) and back to an array of bytes. You may have to convert from bytes to a built-in data type after you read bytes off the network, for example. In addition to the [ToInt32\(Byte\[\], Int32\)](#) method in the example, the following table lists methods in the [BitConverter](#) class that convert bytes (from an array of bytes) to other built-in types.

TYPE RETURNED	METHOD
<code>bool</code>	.ToBoolean(Byte[], Int32)
<code>char</code>	ToChar(Byte[], Int32)
<code>double</code>	.ToDouble(Byte[], Int32)
<code>short</code>	ToInt16(Byte[], Int32)
<code>int</code>	ToInt32(Byte[], Int32)
<code>long</code>	ToInt64(Byte[], Int32)
<code>float</code>	ToSingle(Byte[], Int32)
<code>ushort</code>	ToUInt16(Byte[], Int32)
<code>uint</code>	ToUInt32(Byte[], Int32)
<code>ulong</code>	ToUInt64(Byte[], Int32)

Example

This example initializes an array of bytes, reverses the array if the computer architecture is little-endian (that is, the least significant byte is stored first), and then calls the [ToInt32\(Byte\[\], Int32\)](#) method to convert four bytes in the array to an `int`. The second argument to [ToInt32\(Byte\[\], Int32\)](#) specifies the start index of the array of bytes.

NOTE

The output may differ depending on the endianness of your computer's architecture.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Example

In this example, the [GetBytes\(Int32\)](#) method of the [BitConverter](#) class is called to convert an `int` to an array of bytes.

NOTE

The output may differ depending on the endianness of your computer's architecture.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

See also

- [BitConverter](#)
- [IsLittleEndian](#)
- [Types](#)

How to convert a string to a number (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

You convert a `string` to a number by calling the `Parse` or `TryParse` method found on numeric types (`int`, `long`, `double`, and so on), or by using methods in the `System.Convert` class.

It's slightly more efficient and straightforward to call a `TryParse` method (for example, `int.TryParse("11", out number)`) or `Parse` method (for example, `var number = int.Parse("11")`). Using a `Convert` method is more useful for general objects that implement `IConvertible`.

You use `Parse` or `TryParse` methods on the numeric type you expect the string contains, such as the `System.Int32` type. The `Convert.ToInt32` method uses `Parse` internally. The `Parse` method returns the converted number; the `TryParse` method returns a boolean value that indicates whether the conversion succeeded, and returns the converted number in an `out` parameter. If the string isn't in a valid format, `Parse` throws an exception, but `TryParse` returns `false`. When calling a `Parse` method, you should always use exception handling to catch a `FormatException` when the parse operation fails.

Call Parse or TryParse methods

The `Parse` and `TryParse` methods ignore white space at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (`int`, `long`, `ulong`, `float`, `decimal`, and so on). Any white space within the string that forms the number causes an error. For example, you can use `decimal.TryParse` to parse "10", "10.3", or " 10 ", but you can't use this method to parse 10 from "10X", "1 0" (note the embedded space), "10 .3" (note the embedded space), "10e1" (`float.TryParse` works here), and so on. A string whose value is `null` or `String.Empty` fails to parse successfully. You can check for a null or empty string before attempting to parse it by calling the `String.IsNullOrEmpty` method.

The following example demonstrates both successful and unsuccessful calls to `Parse` and `TryParse`.

```

using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}

```

The following example illustrates one approach to parsing a string expected to include leading numeric characters (including hexadecimal characters) and trailing non-numeric characters. It assigns valid characters from the beginning of a string to a new string before calling the `TryParse` method. Because the strings to be

parsed contain a few characters, the example calls the [String.Concat](#) method to assign valid characters to a new string. For a larger string, the [StringBuilder](#) class can be used instead.

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = " 10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A' && char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: ' 10FFxxx' --> ' 10FF' --> 4351

        str = " -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == '-' || c == ' ')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: ' -10FFXXX' --> ' -10' --> -10
    }
}
```

Call Convert methods

The following table lists some of the methods from the [Convert](#) class that you can use to convert a string to a number.

NUMERIC TYPE	METHOD
decimal	ToDecimal(String)

NUMERIC TYPE	METHOD
<code>float</code>	<code>ToSingle(String)</code>
<code>double</code>	<code>ToDouble(String)</code>
<code>short</code>	<code>ToInt16(String)</code>
<code>int</code>	<code>ToInt32(String)</code>
<code>long</code>	<code>ToInt64(String)</code>
<code>ushort</code>	<code>ToUInt16(String)</code>
<code>uint</code>	<code>ToUInt32(String)</code>
<code>ulong</code>	<code>ToUInt64(String)</code>

The following example calls the [Convert.ToInt32\(String\)](#) method to convert an input string to an `int`. The example catches the two most common exceptions that can be thrown by this method, [FormatException](#) and [OverflowException](#). If the resulting number can be incremented without exceeding [Int32.MaxValue](#), the example adds 1 to the result and displays the output.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.WriteLine("Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            //ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}

// Sample Output:
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
// The new value is 474
// Go again? Y/N: y
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
// numVal cannot be incremented beyond its current value
// Go again? Y/N: y
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
// The new value is -999
// Go again? Y/N: n
```

How to convert between hexadecimal strings and numeric types (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

These examples show you how to perform the following tasks:

- Obtain the hexadecimal value of each character in a [string](#).
- Obtain the [char](#) that corresponds to each value in a hexadecimal string.
- Convert a hexadecimal [string](#) to an [int](#).
- Convert a hexadecimal [string](#) to a [float](#).
- Convert a [byte](#) array to a hexadecimal [string](#).

Example

This example outputs the hexadecimal value of each character in a [string](#). First it parses the [string](#) to an array of characters. Then it calls [ToInt32\(Char\)](#) on each character to obtain its numeric value. Finally, it formats the number as its hexadecimal representation in a [string](#).

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
   Hexadecimal value of H is 48
   Hexadecimal value of e is 65
   Hexadecimal value of l is 6C
   Hexadecimal value of l is 6C
   Hexadecimal value of o is 6F
   Hexadecimal value of   is 20
   Hexadecimal value of W is 57
   Hexadecimal value of o is 6F
   Hexadecimal value of r is 72
   Hexadecimal value of l is 6C
   Hexadecimal value of d is 64
   Hexadecimal value of ! is 21
*/
```

Example

This example parses a [string](#) of hexadecimal values and outputs the character corresponding to each hexadecimal value. First it calls the [Split\(Char\[\]\)](#) method to obtain each hexadecimal value as an individual [string](#) in an array. Then it calls [ToInt32\(String, Int32\)](#) to convert the hexadecimal value to a decimal value represented as an [int](#). It shows two different ways to obtain the character corresponding to that character code. The first technique uses [ConvertFromUtf32\(Int32\)](#), which returns the character corresponding to the integer argument as a [string](#). The second technique explicitly casts the [int](#) to a [char](#).

```

string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
   hexadecimal value = 48, int value = 72, char value = H or h
   hexadecimal value = 65, int value = 101, char value = e or E
   hexadecimal value = 6C, int value = 108, char value = l or L
   hexadecimal value = 6C, int value = 108, char value = l or L
   hexadecimal value = 6F, int value = 111, char value = o or O
   hexadecimal value = 20, int value = 32, char value = @ or @
   hexadecimal value = 57, int value = 87, char value = W or w
   hexadecimal value = 6F, int value = 111, char value = o or O
   hexadecimal value = 72, int value = 114, char value = r or R
   hexadecimal value = 6C, int value = 108, char value = l or L
   hexadecimal value = 64, int value = 100, char value = d or D
   hexadecimal value = 21, int value = 33, char value = ! or !
*/

```

Example

This example shows another way to convert a hexadecimal `string` to an integer, by calling the [Parse\(String, NumberStyles\)](#) method.

```

string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274

```

Example

The following example shows how to convert a hexadecimal `string` to a `float` by using the [System.BitConverter](#) class and the [UInt32.Parse](#) method.

```

string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056

```

Example

The following example shows how to convert a `byte` array to a hexadecimal string by using the [System.BitConverter](#) class.

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
01-AA-B1-DC-10-DD
01AAB1DC10DD
*/
```

Example

The following example shows how to convert a `byte` array to a hexadecimal string by calling the [Convert.ToString](#) method introduced in .NET 5.0.

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
646F74636574
*/
```

See also

- [Standard Numeric Format Strings](#)
- [Types](#)
- [How to determine whether a string represents a numeric value](#)

Using type dynamic (C# Programming Guide)

3/15/2021 • 5 minutes to read • [Edit Online](#)

C# 4 introduces a new type, `dynamic`. The type is a static type, but an object of type `dynamic` bypasses static type checking. In most cases, it functions like it has type `object`. At compile time, an element that is typed as `dynamic` is assumed to support any operation. Therefore, you do not have to be concerned about whether the object gets its value from a COM API, from a dynamic language such as IronPython, from the HTML Document Object Model (DOM), from reflection, or from somewhere else in the program. However, if the code is not valid, errors are caught at run time.

For example, if instance method `exampleMethod1` in the following code has only one parameter, the compiler recognizes that the first call to the method, `ec.exampleMethod1(10, 4)`, is not valid because it contains two arguments. The call causes a compiler error. The second call to the method, `dynamic_ec.exampleMethod1(10, 4)`, is not checked by the compiler because the type of `dynamic_ec` is `dynamic`. Therefore, no compiler error is reported. However, the error does not escape notice indefinitely. It is caught at run time and causes a run-time exception.

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

The role of the compiler in these examples is to package together information about what each statement is proposing to do to the object or expression that is typed as `dynamic`. At run time, the stored information is examined, and any statement that is not valid causes a run-time exception.

The result of most dynamic operations is itself `dynamic`. For example, if you rest the mouse pointer over the use of `testSum` in the following example, IntelliSense displays the type (local variable) `dynamic testSum`.

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Operations in which the result is not `dynamic` include:

- Conversions from `dynamic` to another type.
- Constructor calls that include arguments of type `dynamic`.

For example, the type of `testInstance` in the following declaration is `ExampleClass`, not `dynamic`:

```
var testInstance = new ExampleClass(d);
```

Conversion examples are shown in the following section, "Conversions."

Conversions

Conversions between dynamic objects and other types are easy. This enables the developer to switch between dynamic and non-dynamic behavior.

Any object can be converted to dynamic type implicitly, as shown in the following examples.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Conversely, an implicit conversion can be dynamically applied to any expression of type `dynamic`.

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Overload resolution with arguments of type `dynamic`

Overload resolution occurs at run time instead of at compile time if one or more of the arguments in a method call have the type `dynamic`, or if the receiver of the method call is of type `dynamic`. In the following example, if the only accessible `exampleMethod2` method is defined to take a string argument, sending `d1` as the argument does not cause a compiler error, but it does cause a run-time exception. Overload resolution fails at run time because the run-time type of `d1` is `int`, and `exampleMethod2` requires a string.

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic language runtime

The dynamic language runtime (DLR) is an API that was introduced in .NET Framework 4. It provides the infrastructure that supports the `dynamic` type in C#, and also the implementation of dynamic programming languages such as IronPython and IronRuby. For more information about the DLR, see [Dynamic Language Runtime Overview](#).

COM interop

C# 4 includes several features that improve the experience of interoperating with COM APIs such as the Office Automation APIs. Among the improvements are the use of the `dynamic` type, and of [named and optional arguments](#).

Many COM methods allow for variation in argument types and return type by designating the types as `object`. This has necessitated explicit casting of the values to coordinate with strongly typed variables in C#. If you compile by using the [EmbedInteropTypes \(C# Compiler Options\)](#) option, the introduction of the `dynamic` type enables you to treat the occurrences of `object` in COM signatures as if they were of type `dynamic`, and thereby to avoid much of the casting. For example, the following statements contrast how you access a cell in a Microsoft Office Excel spreadsheet with the `dynamic` type and without the `dynamic` type.

```
// Before the introduction of dynamic.  
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";  
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and  
// the conversion to Excel.Range are handled by the run-time COM binder.  
excelApp.Cells[1, 1].Value = "Name";  
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Related topics

TITLE	DESCRIPTION
dynamic	Describes the usage of the <code>dynamic</code> keyword.
Dynamic Language Runtime Overview	Provides an overview of the DLR, which is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR).
Walkthrough: Creating and Using Dynamic Objects	Provides step-by-step instructions for creating a custom dynamic object and for creating a project that accesses an <code>IronPython</code> library.
How to access Office interop objects by using C# features	Demonstrates how to create a project that uses named and optional arguments, the <code>dynamic</code> type, and other enhancements that simplify access to Office API objects.

Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

3/25/2021 • 11 minutes to read • [Edit Online](#)

Dynamic objects expose members such as properties and methods at run time, instead of at compile time. This enables you to create objects to work with structures that do not match a static type or format. For example, you can use a dynamic object to reference the HTML Document Object Model (DOM), which can contain any combination of valid HTML markup elements and attributes. Because each HTML document is unique, the members for a particular HTML document are determined at run time. A common method to reference an attribute of an HTML element is to pass the name of the attribute to the `GetProperty` method of the element. To reference the `id` attribute of the HTML element `<div id="Div1">`, you first obtain a reference to the `<div>` element, and then use `divElement.GetProperty("id")`. If you use a dynamic object, you can reference the `id` attribute as `divElement.id`.

Dynamic objects also provide convenient access to dynamic languages such as IronPython and IronRuby. You can use a dynamic object to refer to a dynamic script that is interpreted at run time.

You reference a dynamic object by using late binding. In C#, you specify the type of a late-bound object as `dynamic`. In Visual Basic, you specify the type of a late-bound object as `Object`. For more information, see [dynamic](#) and [Early and Late Binding](#).

You can create custom dynamic objects by using the classes in the `System.Dynamic` namespace. For example, you can create an `ExpandoObject` and specify the members of that object at run time. You can also create your own type that inherits the `DynamicObject` class. You can then override the members of the `DynamicObject` class to provide run-time dynamic functionality.

This article contains two independent walkthroughs:

- Create a custom object that dynamically exposes the contents of a text file as properties of an object.
- Create a project that uses an `IronPython` library.

You can do either one of these or both of them, and if you do both, the order doesn't matter.

Prerequisites

- [Visual Studio 2019 version 16.9 or a later version](#) with the **.NET desktop development** workload installed. The .NET 5.0 SDK is automatically installed when you select this workload.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

- For the second walkthrough, install `IronPython` for .NET. Go to their [Download](#) page to obtain the latest version.

Create a Custom Dynamic Object

The first walkthrough defines a custom dynamic object that searches the contents of a text file. A dynamic

property specifies the text to search for. For example, if calling code specifies `dynamicFile.Sample`, the dynamic class returns a generic list of strings that contains all of the lines from the file that begin with "Sample". The search is case-insensitive. The dynamic class also supports two optional arguments. The first argument is a search option enum value that specifies that the dynamic class should search for matches at the start of the line, the end of the line, or anywhere in the line. The second argument specifies that the dynamic class should trim leading and trailing spaces from each line before searching. For example, if calling code specifies

`dynamicFile.Sample(StringSearchOption.Contains)`, the dynamic class searches for "Sample" anywhere in a line. If calling code specifies `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, the dynamic class searches for "Sample" at the start of each line, and does not remove leading and trailing spaces. The default behavior of the dynamic class is to search for a match at the start of each line and to remove leading and trailing spaces.

To create a custom dynamic class

1. Start Visual Studio.
2. Select **Create a new project**.
3. In the **Create a new project** dialog, select C# or Visual Basic, select **Console Application**, and then select **Next**.
4. In the **Configure your new project** dialog, enter `DynamicSample` for the **Project name**, and then select **Next**.
5. In the **Additional information** dialog, select **.NET 5.0 (Current)** for the **Target Framework**, and then select **Create**.

The new project is created.

6. In **Solution Explorer**, right-click the DynamicSample project and select **Add > Class**. In the **Name** box, type `ReadOnlyFile`, and then select **Add**.

A new file is added that contains the `ReadOnlyFile` class.

7. At the top of the `ReadOnlyFile.cs` or `ReadOnlyFile.vb` file, add the following code to import the `System.IO` and `System.Dynamic` namespaces.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

8. The custom dynamic object uses an enum to determine the search criteria. Before the class statement, add the following enum definition.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

9. Update the class statement to inherit the `DynamicObject` class, as shown in the following code example.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

10. Add the following code to the `ReadOnlyFile` class to define a private field for the file path and a constructor for the `ReadOnlyFile` class.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

11. Add the following `GetPropertyValues` method to the `ReadOnlyFile` class. The `GetPropertyValues` method takes, as input, search criteria and returns the lines from a text file that match that search criteria. The dynamic methods provided by the `ReadOnlyFile` class call the `GetPropertyValues` method to retrieve their respective results.

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }
}

return results;
}
```

```

Public Function GetPropertyValue(ByVal propertyName As String,
                               Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                               Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCASE(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCASE(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function

```

12. After the `GetPropertyValue` method, add the following code to override the `TryGetMember` method of the `DynamicObject` class. The `TryGetMember` method is called when a member of a dynamic class is requested and no arguments are specified. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `TryGetMember` method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

```

// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                   out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}

```

```

' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function

```

13. After the `TryGetMember` method, add the following code to override the `TryInvokeMember` method of the `DynamicObject` class. The `TryInvokeMember` method is called when a member of a dynamic class is requested with arguments. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `args` argument contains an array of the arguments that are passed to the member. The `TryInvokeMember` method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

The custom version of the `TryInvokeMember` method expects the first argument to be a value from the `StringSearchOption` enum that you defined in a previous step. The `TryInvokeMember` method expects the second argument to be a Boolean value. If one or both arguments are valid values, they are passed to the `GetPropertyValues` method to retrieve the results.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                      object[] args,
                                      out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```

' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                         ByVal args() As Object,
                                         ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function

```

14. Save and close the file.

To create a sample text file

1. In **Solution Explorer**, right-click the **DynamicSample** project and select **Add > New Item**. In the **Installed Templates** pane, select **General**, and then select the **Text File** template. Leave the default name of *TextFile1.txt* in the **Name** box, and then click **Add**. A new text file is added to the project.
2. Copy the following text to the *TextFile1.txt* file.

```

List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul

```

3. Save and close the file.

To create a sample application that uses the custom dynamic object

1. In **Solution Explorer**, double-click the *Program.vb* file if you're using Visual Basic or the *Program.cs* file if you're using Visual C#.
2. Add the following code to the `Main` procedure to create an instance of the `ReadOnlyFile` class for the *TextFile1.txt* file. The code uses late binding to call dynamic members and retrieve lines of text that contain the string "Customer".

```

dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("-----");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}

```

```

Dim rFile As Object = New ReadOnlyFile("../..\\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("-----")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next

```

- Save the file and press **Ctrl+F5** to build and run the application.

Call a dynamic language library

The following walkthrough creates a project that accesses a library that is written in the dynamic language IronPython.

To create a custom dynamic class

- In Visual Studio, select **File > New > Project**.
- In the **Create a new project** dialog, select C# or Visual Basic, select **Console Application**, and then select **Next**.
- In the **Configure your new project** dialog, enter **DynamicIronPythonSample** for the **Project name**, and then select **Next**.
- In the **Additional information** dialog, select **.NET 5.0 (Current)** for the **Target Framework**, and then select **Create**.

The new project is created.

- Install the [IronPython](#) NuGet package.
- If you're using Visual Basic, edit the *Program.vb* file. If you're using Visual C#, edit the *Program.cs* file.
- At the top of the file, add the following code to import the **Microsoft.Scripting.Hosting** and **IronPython.Hosting** namespaces from the IronPython libraries and the **System.Linq** namespace.

```

using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;

```

```

Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
Imports System.Linq

```

- In the **Main** method, add the following code to create a new **Microsoft.Scripting.Hosting.ScriptRuntime**

object to host the IronPython libraries. The `ScriptRuntime` object loads the IronPython library module `random.py`.

```
// Set the current directory to the IronPython libraries.  
System.IO.Directory.SetCurrentDirectory(  
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +  
    @"\IronPython 2.7\Lib");  
  
// Create an instance of the random.py IronPython library.  
Console.WriteLine("Loading random.py");  
ScriptRuntime py = Python.CreateRuntime();  
dynamic random = py.UseFile("random.py");  
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.  
System.IO.Directory.SetCurrentDirectory(  
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) &  
    "\IronPython 2.7\Lib")  
  
' Create an instance of the random.py IronPython library.  
Console.WriteLine("Loading random.py")  
Dim py = Python.CreateRuntime()  
Dim random As Object = py.UseFile("random.py")  
Console.WriteLine("random.py loaded.")
```

9. After the code to load the `random.py` module, add the following code to create an array of integers. The array is passed to the `shuffle` method of the `random.py` module, which randomly sorts the values in the array.

```
// Initialize an enumerable set of integers.  
int[] items = Enumerable.Range(1, 7).ToArray();  
  
// Randomly shuffle the array of integers by using IronPython.  
for (int i = 0; i < 5; i++)  
{  
    random.shuffle(items);  
    foreach (int item in items)  
    {  
        Console.WriteLine(item);  
    }  
    Console.WriteLine("-----");  
}
```

```
' Initialize an enumerable set of integers.  
Dim items = Enumerable.Range(1, 7).ToArray()  
  
' Randomly shuffle the array of integers by using IronPython.  
For i = 0 To 4  
    random.shuffle(items)  
    For Each item In items  
        Console.WriteLine(item)  
    Next  
    Console.WriteLine("-----")  
Next
```

10. Save the file and press **Ctrl+F5** to build and run the application.

See also

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [Using Type dynamic](#)
- [Early and Late Binding](#)
- [dynamic](#)
- [Implementing Dynamic Interfaces \(downloadable PDF from Microsoft TechNet\)](#)

Classes, structs, and records (C# programming guide)

4/7/2021 • 7 minutes to read • [Edit Online](#)

Classes and structs are two of the basic constructs of the common type system in .NET. C# 9 adds records, which are a kind of class. Each is essentially a data structure that encapsulates a set of data and behaviors that belong together as a logical unit. The data and behaviors are the *members* of the class, struct, or record, and they include its methods, properties, events, and so on, as listed later in this article.

A class, struct, or record declaration is like a blueprint that is used to create instances or objects at run time. If you define a class, struct, or record named `Person`, `Person` is the name of the type. If you declare and initialize a variable `p` of type `Person`, `p` is said to be an object or instance of `Person`. Multiple instances of the same `Person` type can be created, and each instance can have different values in its properties and fields.

A class or a record is a reference type. When an object of the type is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

A struct is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

In general, classes are used to model more complex behavior, or data that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that isn't intended to be modified after the struct is created. Record types are for larger data structures that contain primarily data that isn't intended to be modified after the object is created.

Example

In the following example, `CustomClass` in the `ProgrammingGuide` namespace has three members: an instance constructor, a property named `Number`, and a method named `Multiply`. The `Main` method in the `Program` class creates an instance (object) of `CustomClass`, and the object's method and property are accessed by using dot notation.

```

using System;

namespace ProgrammingGuide
{
    // Class definition.
    public class CustomClass
    {
        // Class members.
        //
        // Property.
        public int Number { get; set; }

        // Method.
        public int Multiply(int num)
        {
            return num * Number;
        }

        // Instance Constructor.
        public CustomClass()
        {
            Number = 0;
        }
    }

    // Another class definition that contains Main, the program entry point.
    class Program
    {
        static void Main(string[] args)
        {
            // Create an object of type CustomClass.
            CustomClass custClass = new CustomClass();

            // Set the value of the public property.
            custClass.Number = 27;

            // Call the public method.
            int result = custClass.Multiply(4);
            Console.WriteLine($"The result is {result}.");
        }
    }
}

// The example displays the following output:
//      The result is 108.

```

Encapsulation

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. According to the principle of encapsulation, a class or struct can specify how accessible each of its members is to code outside of the class or struct. Methods and variables that are not intended to be used from outside of the class or assembly can be hidden to limit the potential for coding errors or malicious exploits. For more information, see [Object-oriented programming](#).

Members

All methods, fields, constants, properties, and events must be declared within a type; these are called the *members* of the type. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the `Main` method, must be declared within a class or struct (implicitly in the case of [top-level statements](#)).

The following list includes all the various kinds of members that may be declared in a class, struct, or record.

- [Fields](#)
- [Constants](#)
- [Properties](#)
- [Methods](#)
- [Constructors](#)
- [Events](#)
- [Finalizers](#)
- [Indexers](#)
- [Operators](#)
- [Nested Types](#)

Accessibility

Some methods and properties are meant to be called or accessed from code outside a class or struct, known as *client code*. Other methods and properties might be only for use in the class or struct itself. It is important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the following access modifiers:

- [public](#)
- [protected](#)
- [internal](#)
- [protected internal](#)
- [private](#)
- [private protected](#).

The default accessibility is `private`. For more information, see [Access Modifiers](#).

Inheritance

Classes (but not structs) support the concept of inheritance. A class that derives from another class (the *base class*) automatically contains all the public, protected, and internal members of the base class except its constructors and finalizers. For more information, see [Inheritance](#) and [Polymorphism](#).

Classes may be declared as [abstract](#), which means that one or more of their methods have no implementation. Although abstract classes cannot be instantiated directly, they can serve as base classes for other classes that provide the missing implementation. Classes can also be declared as [sealed](#) to prevent other classes from inheriting from them. For more information, see [Abstract and sealed classes and class members](#).

Interfaces

Classes, structs, and records can inherit multiple interfaces. To inherit from an interface means that the type implements all the methods defined in the interface. For more information, see [Interfaces](#).

Generic Types

Classes, structs, and records can be defined with one or more type parameters. Client code supplies the type when it creates an instance of the type. For example The `List<T>` class in the `System.Collections.Generic` namespace is defined with one type parameter. Client code creates an instance of a `List<string>` or `List<int>` to specify the type that the list will hold. For more information, see [Generics](#).

Static Types

Classes (but not structs or records) can be declared as [static](#). A static class can contain only static members and can't be instantiated with the `new` keyword. One copy of the class is loaded into memory when the program loads, and its members are accessed through the class name. Classes, structs, and records can contain static members. For more information, see [Static classes and static class members](#).

Nested Types

A class, struct, or record can be nested within another class, struct, or record. For more information, see [Nested Types](#).

Partial Types

You can define part of a class, struct or method in one code file and another part in a separate code file. For more information, see [Partial Classes and methods](#).

Object Initializers

You can instantiate and initialize class or struct objects, and collections of objects, without explicitly calling their constructor. For more information, see [Object and collection initializers](#).

Anonymous Types

In situations where it is not convenient or necessary to create a named class, for example when you are populating a list with data structures that you do not have to persist or pass to another method, you use anonymous types. For more information, see [Anonymous types](#).

Extension Methods

You can "extend" a class without creating a derived class by creating a separate type whose methods can be called as if they belonged to the original type. For more information, see [Extension methods](#).

Implicitly Typed Local Variables

Within a class or struct method, you can use implicit typing to instruct the compiler to determine a variable's type at compile time. For more information, see [Implicitly Typed Local Variables](#).

Records

C# 9 introduces the `record` type, a reference type that you can create instead of a class or a struct. Records are classes with built-in behavior for encapsulating data in immutable types. A record provides the following features:

- Concise syntax for creating a reference type with immutable properties.
- Value equality.

Two variables of a record type are equal if the record type definitions are identical, and if for every field, the values in both records are equal. This differs from classes, which use reference equality: two variables of a class type are equal if they refer to the same object.

- Concise syntax for nondestructive mutation.

A `with` expression lets you create a new record instance that is a copy of an existing instance but with specified property values changed.

- Built-in formatting for display.

The `ToString` method prints out the record type name and the names and values of public properties.

- Support for inheritance hierarchies.

Inheritance is supported because a record is a class under the covers, not a struct.

For more information, see [Records](#).

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Classes](#)
- [Objects](#)
- [Structure types](#)
- [Records](#)
- [C# Programming Guide](#)

Classes (C# Programming Guide)

3/6/2021 • 5 minutes to read • [Edit Online](#)

Reference types

A type that is defined as a [class](#) is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value [null](#) until you explicitly create an instance of the class by using the [new](#) operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

```
//Declaring an object of type MyClass.  
MyClass mc = new MyClass();  
  
//Declaring another object of the same type, assigning it the value of the first object.  
MyClass mc2 = mc;
```

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized and in most scenarios, it does not create a performance issue. For more information about garbage collection, see [Automatic memory management and garbage collection](#).

Declaring Classes

Classes are declared by using the [class](#) keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]  
public class Customer  
{  
    // Fields, properties, methods and events go here...  
}
```

The [class](#) keyword is preceded by the access level. Because [public](#) is used in this case, anyone can create instances of this class. The name of the class follows the [class](#) keyword. The name of the class must be a valid C# [identifier name](#). The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the [new](#) keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous example, `object1` is a reference to an object that is based on `Customer`. This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

We don't recommend creating object references such as this one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it an existing object, such as this:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through `object3` are reflected in subsequent uses of `object4`. Because objects that are based on classes are referred to by reference, classes are known as reference types.

Class inheritance

Classes fully support *inheritance*, a fundamental characteristic of object-oriented programming. When you create a class, you can inherit from any other class that is not defined as `sealed`, and other classes can inherit from your class and override class virtual methods. Furthermore, you can implement one or more interfaces.

Inheritance is accomplished by using a *derivation*, which means a class is declared by using a *base class* from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see [Inheritance](#).

Unlike C++, a class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement one or more interfaces. For more information, see [Interfaces](#).

A class can be declared `abstract`. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a `sealed` class does not allow other classes to derive from it. For more information, see [Abstract and Sealed Classes and Class Members](#).

Class definitions can be split between different source files. For more information, see [Partial Classes and Methods](#).

Example

The following example defines a public class that contains an [auto-implemented property](#), a method, and a special method called a constructor. For more information, see [Properties, Methods](#), and [Constructors](#) topics. The instances of the class are then instantiated with the `new` keyword.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Object-Oriented Programming](#)
- [Polymorphism](#)
- [Identifier names](#)
- [Members](#)
- [Methods](#)

- Constructors
- Finalizers
- Objects

Records (C# Programming Guide)

4/3/2021 • 3 minutes to read • [Edit Online](#)

A [record](#) is a [class](#) that provides special syntax and behavior for working with data models. For information about classes, see [Classes \(C# Programming Guide\)](#).

When to use records

Consider using a record in place of a class in the following scenarios:

- You want to define a reference type for which objects are immutable.
- You want to define a data model that depends on [value equality](#).

Immutability

An immutable type is one that prevents you from changing any property or field values of an object after it's instantiated. Immutability can be useful when you need a type to be thread-safe or you're depending on a hash code remaining the same in a hash table. Records provide concise syntax for creating and working with immutable types.

Immutability isn't appropriate for all data scenarios. [Entity Framework Core](#), for example, doesn't support updating with immutable entity types.

Value equality

For records, value equality means that two variables of a record type are equal if the types match and all property and field values match. For other reference types such as classes, equality means [reference equality](#). That is, two variables of a class type are equal if they refer to the same object. Methods and operators that determine equality of two record instances use value equality.

Not all data models work well with value equality. For example, [Entity Framework Core](#) depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, record types aren't appropriate for use as entity types in Entity Framework Core.

How records differ from classes

The same syntax that [declares](#) and [instantiates](#) classes can be used with records. Just substitute the `record` keyword for the `class` keyword. Likewise, the same syntax for expressing inheritance relationships is supported by records. Records differ from classes in the following ways:

- You can use [positional parameters](#) to create and instantiate a type with immutable properties.
- The same methods and operators that indicate reference equality or inequality in classes (such as `Object.Equals(Object)` and `==`), indicate [value equality or inequality](#) in records.
- You can use a [with expression](#) to create a copy of an immutable object with new values in selected properties.
- A record's `ToString` method creates a formatted string that shows an object's type name and the names and values of all its public properties.
- A record can [inherit from another record](#). A record can't inherit from a class, and a class can't inherit from a record.

Examples

The following example defines a public record that uses positional parameters to declare and instantiate a record. It then prints out the type name and property values:

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

The following example demonstrates value equality in records:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

The following example demonstrates use of a `with` expression to copy an immutable object and change one of the properties:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

For more information, see [Records \(C# reference\)](#).

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Classes \(C# Programming Guide\)](#)
- [Records \(C# reference\)](#)
- [C# Programming Guide](#)
- [Object-Oriented Programming](#)
- [Polymorphism](#)
- [Identifier names](#)
- [Members](#)
- [Methods](#)
- [Constructors](#)
- [Finalizers](#)
- [Objects](#)

Objects (C# Programming Guide)

3/6/2021 • 5 minutes to read • [Edit Online](#)

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically.

NOTE

Static types behave differently than what is described here. For more information, see [Static Classes and Static Class Members](#).

Struct Instances vs. Class Instances

Because classes are reference types, a variable of a class object holds a reference to the address of the object on the managed heap. If a second object of the same type is assigned to the first object, then both variables refer to the object at that address. This point is discussed in more detail later in this topic.

Instances of classes are created by using the [new operator](#). In the following example, `Person` is the type and `person1` and `person2` are instances, or objects, of that type.

```

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
person1 Name = Leopold Age = 6
person2 Name = Molly Age = 16
person1 Name = Molly Age = 16
*/

```

Because structs are value types, a variable of a struct object holds a copy of the entire object. Instances of structs can also be created by using the `new` operator, but this is not required, as shown in the following example:

```

public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create new struct object. Note that struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
p1 Name = Alex Age = 9
p2 Name = Spencer Age = 7
p1 Name = Alex Age = 9
*/

```

The memory for both `p1` and `p2` is allocated on the thread stack. That memory is reclaimed along with the type or method in which it is declared. This is one reason why structs are copied on assignment. By contrast, the memory that is allocated for a class instance is automatically reclaimed (garbage collected) by the common language runtime when all references to the object have gone out of scope. It is not possible to deterministically destroy a class object like you can in C++. For more information about garbage collection in .NET, see [Garbage Collection](#).

NOTE

The allocation and deallocation of memory on the managed heap is highly optimized in the common language runtime. In most cases there is no significant difference in the performance cost of allocating a class instance on the heap versus allocating a struct instance on the stack.

Object Identity vs. Value Equality

When you compare two objects for equality, you must first distinguish whether you want to know whether the two variables represent the same object in memory, or whether the values of one or more of their fields are

equivalent. If you are intending to compare values, you must consider whether the objects are instances of value types (structs) or reference types (classes, delegates, arrays).

- To determine whether two class instances refer to the same location in memory (which means that they have the same *identity*), use the static `Equals` method. (`System.Object` is the implicit base class for all value types and reference types, including user-defined structs and classes.)
- To determine whether the instance fields in two struct instances have the same values, use the `ValueType.Equals` method. Because all structs implicitly inherit from `System.ValueType`, you call the method directly on your object as shown in the following example:

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

The `System.ValueType` implementation of `Equals` uses boxing and reflection in some cases. For information about how to provide an efficient equality algorithm that is specific to your type, see [How to define value equality for a type](#)

- To determine whether the values of the fields in two class instances are equal, you might be able to use the `Equals` method or the `== operator`. However, only use them if the class has overridden or overloaded them to provide a custom definition of what "equality" means for objects of that type. The class might also implement the `IEquatable<T>` interface or the `IEqualityComparer<T>` interface. Both interfaces provide methods that can be used to test value equality. When designing your own classes that override `Equals`, make sure to follow the guidelines stated in [How to define value equality for a type](#) and [Object.Equals\(Object\)](#).

Related Sections

For more information:

- [Classes](#)
- [Constructors](#)
- [Finalizers](#)
- [Events](#)

See also

- [C# Programming Guide](#)

- [object](#)
- [Inheritance](#)
- [class](#)
- [Structure types](#)
- [new Operator](#)
- [Common Type System](#)

Inheritance (C# Programming Guide)

11/2/2020 • 7 minutes to read • [Edit Online](#)

Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior defined in other classes. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. A derived class can have only one direct base class. However, inheritance is transitive. If `ClassC` is derived from `ClassB`, and `ClassB` is derived from `ClassA`, `ClassC` inherits the members declared in `ClassB` and `ClassA`.

NOTE

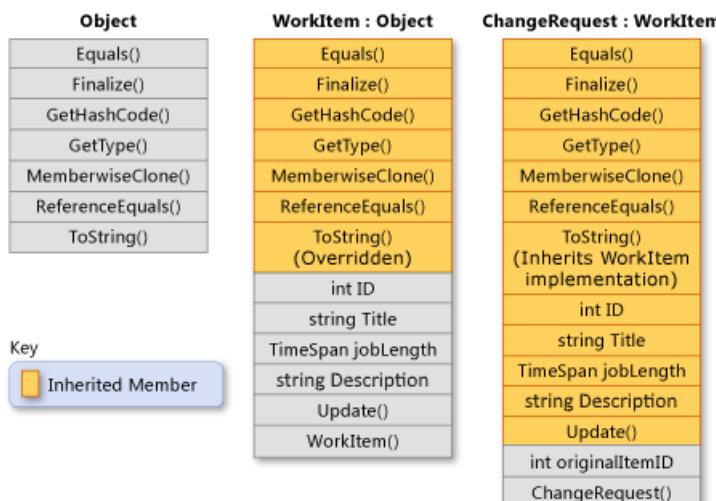
Structs do not support inheritance, but they can implement interfaces. For more information, see [Interfaces](#).

Conceptually, a derived class is a specialization of the base class. For example, if you have a base class `Animal`, you might have one derived class that is named `Mammal` and another derived class that is named `Reptile`. A `Mammal` is an `Animal`, and a `Reptile` is an `Animal`, but each derived class represents different specializations of the base class.

Interface declarations may define a default implementation for its members. These implementations are inherited by derived interfaces, and by classes that implement those interfaces. For more information on default interface methods, see the article on [interfaces](#) in the language reference section.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class reuses the code in the base class without having to reimplement it. You can add more members in the derived class. The derived class extends the functionality of the base class.

The following illustration shows a class `WorkItem` that represents an item of work in some business process. Like all classes, it derives from `System.Object` and inherits all its methods. `WorkItem` adds five members of its own. These members include a constructor, because constructors aren't inherited. Class `ChangeRequest` inherits from `WorkItem` and represents a particular kind of work item. `ChangeRequest` adds two more members to the members that it inherits from `WorkItem` and from `Object`. It must add its own constructor, and it also adds `originalItemID`. Property `originalItemID` enables the `ChangeRequest` instance to be associated with the original `WorkItem` to which the change request applies.



The following example shows how the class relationships demonstrated in the previous illustration are expressed in C#. The example also shows how `WorkItem` overrides the virtual method `Object.ToString`, and how the `ChangeRequest` class inherits the `WorkItem` implementation of the method. The first block defines the classes:

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
```

```

// constructor explicitly, the default constructor in the base class
// is called implicitly. The base class must contain a default
// constructor.

// Default constructor for the derived class.
public ChangeRequest() { }

// Instance constructor that has four parameters.
public ChangeRequest(string title, string desc, TimeSpan jobLen,
                     int originalID)
{
    // The following properties and the GetNextID method are inherited
    // from WorkItem.
    this.ID = GetNextID();
    this.Title = title;
    this.Description = desc;
    this.jobLength = jobLen;

    // Property originalItemId is a member of ChangeRequest, but not
    // of WorkItem.
    this.originalItemId = originalID;
}
}

```

This next block shows how to use the base and derived classes:

```

// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                            "Fix all bugs in my code branch",
                            new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
             new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
   1 - Fix Bugs
   2 - Change the Design of the Base Class
*/

```

Abstract and virtual methods

When a base class declares a method as `virtual`, a derived class can `override` the method with its own implementation. If a base class declares a member as `abstract`, that method must be overridden in any non-abstract class that directly inherits from that class. If a derived class is itself abstract, it inherits abstract members without implementing them. Abstract and virtual members are the basis for polymorphism, which is the second primary characteristic of object-oriented programming. For more information, see [Polymorphism](#).

Abstract base classes

You can declare a class as [abstract](#) if you want to prevent direct instantiation by using the [new](#) operator. An abstract class can be used only if a new class is derived from it. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract class doesn't have to contain abstract members; however, if a class does contain an abstract member, the class itself must be declared as abstract. Derived classes that aren't abstract themselves must provide the implementation for any abstract methods from an abstract base class. For more information, see [Abstract and Sealed Classes and Class Members](#).

Interfaces

An *interface* is a reference type that defines a set of members. All classes and structs that implement that interface must implement that set of members. An interface may define a default implementation for any or all of these members. A class can implement multiple interfaces even though it can derive from only a single direct base class.

Interfaces are used to define specific capabilities for classes that don't necessarily have an "is a" relationship. For example, the [System.IEquatable<T>](#) interface can be implemented by any class or struct to determine whether two objects of the type are equivalent (however the type defines equivalence). [IEquatable<T>](#) doesn't imply the same kind of "is a" relationship that exists between a base class and a derived class (for example, a [Mammal](#) is an [Animal](#)). For more information, see [Interfaces](#).

Preventing further derivation

A class can prevent other classes from inheriting from it, or from any of its members, by declaring itself or the member as [sealed](#). For more information, see [Abstract and Sealed Classes and Class Members](#).

Derived class hiding of base class members

A derived class can hide base class members by declaring members with the same name and signature. The [new](#) modifier can be used to explicitly indicate that the member isn't intended to be an override of the base member. The use of [new](#) isn't required, but a compiler warning will be generated if [new](#) isn't used. For more information, see [Versioning with the Override and New Keywords](#) and [Knowing When to Use Override and New Keywords](#).

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [class](#)

Polymorphism (C# Programming Guide)

11/2/2020 • 7 minutes to read • [Edit Online](#)

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement *virtual methods*, and derived classes can *override* them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called `Shape`, and derived classes such as `Rectangle`, `Circle`, and `Triangle`. Give the `Shape` class a virtual method called `Draw`, and override it in each derived class to draw the particular shape that the class represents. Create a `List<Shape>` object and add a `Circle`, `Triangle`, and `Rectangle` to it.

```

public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}

public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

```

To update the drawing surface, use a `foreach` loop to iterate through the list and call the `Draw` method on each `Shape` object in the list. Even though each object in the list has a declared type of `Shape`, it's the run-time type (the overridden version of the method in each derived class) that will be invoked.

```

// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
   Drawing a rectangle
   Performing base class drawing tasks
   Drawing a triangle
   Performing base class drawing tasks
   Drawing a circle
   Performing base class drawing tasks
*/

```

In C#, every type is polymorphic because all types, including user-defined types, inherit from [Object](#).

Polymorphism overview

Virtual members

When a derived class inherits from a base class, it gains all the methods, fields, properties, and events of the base class. The designer of the derived class has different choices for the behavior of virtual methods:

- The derived class may override virtual members in the base class, defining new behavior.
- The derived class inherit the closest base class method without overriding it, preserving the existing behavior but enabling further derived classes to override the method.
- The derived class may define new non-virtual implementation of those members that hide the base class implementations.

A derived class can override a base class member only if the base class member is declared as [virtual](#) or [abstract](#). The derived member must use the [override](#) keyword to explicitly indicate that the method is intended to participate in virtual invocation. The following code provides an example:

```

public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}

```

Fields cannot be virtual; only methods, properties, events, and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. The following code provides an example:

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork(); // Also calls the new method.
```

Virtual methods and properties enable derived classes to extend a base class without needing to use the base class implementation of a method. For more information, see [Versioning with the Override and New Keywords](#). An interface provides another way to define a method or set of methods whose implementation is left to derived classes. For more information, see [Interfaces](#).

Hide base class members with new members

If you want your derived class to have a member with the same name as a member in a base class, you can use the `new` keyword to hide the base class member. The `new` keyword is put before the return type of a class member that is being replaced. The following code provides an example:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Hidden base class members may be accessed from client code by casting the instance of the derived class to an instance of the base class. For example:

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork(); // Calls the old method.
```

Prevent derived classes from overriding virtual members

Virtual members remain virtual, regardless of how many classes have been declared between the virtual member and the class that originally declared it. If class `A` declares a virtual member, and class `B` derives from `A`, and class `C` derives from `B`, class `C` inherits the virtual member, and may override it, regardless of whether class `B` declared an override for that member. The following code provides an example:

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as [sealed](#). Stopping inheritance requires putting the `sealed` keyword before the `override` keyword in the class member declaration. The following code provides an example:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

In the previous example, the method `DoWork` is no longer virtual to any class derived from `C`. It's still virtual for instances of `C`, even if they're cast to type `B` or type `A`. Sealed methods can be replaced by derived classes by using the `new` keyword, as the following example shows:

```
public class D : C
{
    public new void DoWork() { }
}
```

In this case, if `DoWork` is called on `D` using a variable of type `D`, the new `DoWork` is called. If a variable of type `C`, `B`, or `A` is used to access an instance of `D`, a call to `DoWork` will follow the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class `C`.

Access base class virtual members from derived classes

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the `base` keyword. The following code provides an example:

```
public class Base
{
    public virtual void DoWork() /*...*/
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

For more information, see [base](#).

NOTE

It is recommended that virtual members use `base` to call the base class implementation of that member in their own implementation. Letting the base class behavior occur enables the derived class to concentrate on implementing behavior specific to the derived class. If the base class implementation is not called, it is up to the derived class to make their behavior compatible with the behavior of the base class.

In this section

- [Versioning with the Override and New Keywords](#)
- [Knowing When to Use Override and New Keywords](#)
- [How to override the `ToString` method](#)

See also

- [C# Programming Guide](#)
- [Inheritance](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Methods](#)
- [Events](#)
- [Properties](#)
- [Indexers](#)
- [Types](#)

Versioning with the Override and New Keywords (C# Programming Guide)

11/2/2020 • 5 minutes to read • [Edit Online](#)

The C# language is designed so that versioning between `base` and derived classes in different libraries can evolve and maintain backward compatibility. This means, for example, that the introduction of a new member in a base `class` with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that hides a similarly named inherited method.

In C#, derived classes can contain methods with the same name as base class methods.

- If the method in the derived class is not preceded by `new` or `override` keywords, the compiler will issue a warning and the method will behave as if the `new` keyword were present.
- If the method in the derived class is preceded with the `new` keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the `override` keyword, objects of the derived class will call that method instead of the base class method.
- In order to apply the `override` keyword to the method in the derived class, the base class method must be defined `virtual`.
- The base class method can be called from within the derived class using the `base` keyword.
- The `override`, `virtual`, and `new` keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the `virtual` modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the `override` keyword or hide the virtual method in the base class by using the `new` keyword. If neither the `override` keyword nor the `new` keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.

To demonstrate this in practice, assume for a moment that Company A has created a class named `GraphicsClass`, which your program uses. The following is `GraphicsClass`:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Your company uses this class, and you use it to derive your own class, adding a new method:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Your application is used without problems, until Company A releases a new version of `GraphicsClass`, which resembles the following code:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

The new version of `GraphicsClass` now contains a method named `DrawRectangle`. Initially, nothing occurs. The new version is still binary compatible with the old version. Any software that you have deployed will continue to work, even if the new class is installed on those computer systems. Any existing calls to the method `DrawRectangle` will continue to reference your version, in your derived class.

However, as soon as you recompile your application by using the new version of `GraphicsClass`, you will receive a warning from the compiler, CS0108. This warning informs you that you have to consider how you want your `DrawRectangle` method to behave in your application.

If you want your method to override the new base class method, use the `override` keyword:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

The `override` keyword makes sure that any objects derived from `YourDerivedGraphicsClass` will use the derived class version of `DrawRectangle`. Objects derived from `YourDerivedGraphicsClass` can still access the base class version of `DrawRectangle` by using the `base` keyword:

```
base.DrawRectangle();
```

If you do not want your method to override the new base class method, the following considerations apply. To avoid confusion between the two methods, you can rename your method. This can be time-consuming and error-prone, and just not practical in some cases. However, if your project is relatively small, you can use Visual Studio's Refactoring options to rename the method. For more information, see [Refactoring Classes and Types \(Class Designer\)](#).

Alternatively, you can prevent the warning by using the keyword `new` in your derived class definition:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Using the `new` keyword tells the compiler that your definition hides the definition that is contained in the base class. This is the default behavior.

Override and Method Selection

When a method is named on a class, the C# compiler selects the best method to call if more than one method is compatible with the call, such as when there are two methods with the same name, and parameters that are compatible with the parameter passed. The following methods would be compatible:

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

When `DoWork` is called on an instance of `Derived`, the C# compiler will first try to make the call compatible with the versions of `DoWork` declared originally on `Derived`. Override methods are not considered as declared on a class, they are new implementations of a method declared on a base class. Only if the C# compiler cannot match the method call to an original method on `Derived` will it try to match the call to an overridden method with the same name and compatible parameters. For example:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val); // Calls DoWork(double).
```

Because the variable `val` can be converted to a double implicitly, the C# compiler calls `DoWork(double)` instead of `DoWork(int)`. There are two ways to avoid this. First, avoid declaring new methods with the same name as virtual methods. Second, you can instruct the C# compiler to call the virtual method by making it search the base class method list by casting the instance of `Derived` to `Base`. Because the method is virtual, the implementation of `DoWork(int)` on `Derived` will be called. For example:

```
((Base)d).DoWork(val); // Calls DoWork(int) on Derived.
```

For more examples of `new` and `override`, see [Knowing When to Use Override and New Keywords](#).

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Methods](#)
- [Inheritance](#)

Knowing When to Use Override and New Keywords (C# Programming Guide)

11/2/2020 • 10 minutes to read • [Edit Online](#)

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the `new` and `override` keywords. The `override` modifier *extends* the base class `virtual` method, and the `new` modifier *hides* an accessible base class method. The difference is illustrated in the examples in this topic.

In a console application, declare the following two classes, `BaseClass` and `DerivedClass`. `DerivedClass` inherits from `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the `Main` method, declare variables `bc`, `dc`, and `bcdc`.

- `bc` is of type `BaseClass`, and its value is of type `BaseClass`.
- `dc` is of type `DerivedClass`, and its value is of type `DerivedClass`.
- `bcdc` is of type `BaseClass`, and its value is of type `DerivedClass`. This is the variable to pay attention to.

Because `bc` and `bcdc` have type `BaseClass`, they can only directly access `Method1`, unless you use casting. Variable `dc` can access both `Method1` and `Method2`. These relationships are shown in the following code.

```

class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}

```

Next, add the following `Method2` method to `BaseClass`. The signature of this method matches the signature of the `Method2` method in `DerivedClass`.

```

public void Method2()
{
    Console.WriteLine("Base - Method2");
}

```

Because `BaseClass` now has a `Method2` method, a second calling statement can be added for `BaseClass` variables `bc` and `bcdc`, as shown in the following code.

```

bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();

```

When you build the project, you see that the addition of the `Method2` method in `BaseClass` causes a warning. The warning says that the `Method2` method in `DerivedClass` hides the `Method2` method in `BaseClass`. You are advised to use the `new` keyword in the `Method2` definition if you intend to cause that result. Alternatively, you could rename one of the `Method2` methods to resolve the warning, but that is not always practical.

Before adding `new`, run the program to see the output produced by the additional calling statements. The following results are displayed.

```

// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2

```

The `new` keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type `BaseClass` continue to access the members of `BaseClass`, and the variable that has type `DerivedClass` continues to access members in `DerivedClass` first, and then to consider members inherited

```
from BaseClass .
```

To suppress the warning, add the `new` modifier to the definition of `Method2` in `DerivedClass`, as shown in the following code. The modifier can be added before or after `public`.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using `new`, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance, see [new Modifier](#).

To contrast this behavior to the effects of using `override`, add the following method to `DerivedClass`. The `override` modifier can be added before or after `public`.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the `virtual` modifier to the definition of `Method1` in `BaseClass`. The `virtual` modifier can be added before or after `public`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the `override` modifier enables `bcdc` to access the `Method1` method that is defined in `DerivedClass`. Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using `override` to extend the base class method.

The following code contains the full example.

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

The following example illustrates similar behavior in a different context. The example defines three classes: a base class named `Car` and two classes that are derived from it, `ConvertibleCar` and `Minivan`. The base class contains a `DescribeCar` method. The method displays a basic description of a car, and then calls `ShowDetails` to provide additional information. Each of the three classes defines a `ShowDetails` method. The `new` modifier is used to define `ShowDetails` in the `ConvertibleCar` class. The `override` modifier is used to define `ShowDetails` in the `Minivan` class.

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

The example tests which version of `ShowDetails` is called. The following method, `TestCars1`, declares an instance of each class, and then calls `DescribeCar` on each instance.

```

public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("-----");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("-----");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("-----");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("-----");
}

```

`TestCars1` produces the following output. Notice especially the results for `car2`, which probably are not what you expected. The type of the object is `ConvertibleCar`, but `DescribeCar` does not access the version of `ShowDetails` that is defined in the `ConvertibleCar` class because that method is declared with the `new` modifier, not the `override` modifier. As a result, a `ConvertibleCar` object displays the same description as a `Car` object. Contrast the results for `car3`, which is a `Minivan` object. In this case, the `ShowDetails` method that is declared in the `Minivan` class overrides the `showDetails` method that is declared in the `Car` class, and the description that is displayed describes a minivan.

```

// TestCars1
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----

```

`TestCars2` creates a list of objects that have type `Car`. The values of the objects are instantiated from the `Car`, `ConvertibleCar`, and `Minivan` classes. `DescribeCar` is called on each element of the list. The following code shows the definition of `TestCars2`.

```

public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("-----");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("-----");
    }
}

```

The following output is displayed. Notice that it is the same as the output that is displayed by `TestCars1`. The `ShowDetails` method of the `ConvertibleCar` class is not called, regardless of whether the type of the object is `ConvertibleCar`, as in `TestCars1`, or `Car`, as in `TestCars2`. Conversely, `car3` calls the `ShowDetails` method from the `Minivan` class in both cases, whether it has type `Minivan` or type `Car`.

```
// TestCars2
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----
```

Methods `TestCars3` and `TestCars4` complete the example. These methods call `ShowDetails` directly, first from objects declared to have type `ConvertibleCar` and `Minivan` (`TestCars3`), then from objects declared to have type `Car` (`TestCars4`). The following code defines these two methods.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("-----");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("-----");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

The methods produce the following output, which corresponds to the results from the first example in this topic.

```
// TestCars3
// -----
// A roof that opens up.
// Carries seven people.

// TestCars4
// -----
// Standard transportation.
// Carries seven people.
```

The following code shows the complete project and its output.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

namespace overrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("-----");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("-----");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("-----");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("-----");
        }
        // Output:
        // TestCars1
        // -----
        // Four wheels and an engine.
        // Standard transportation.
        // -----
        // Four wheels and an engine.
        // Standard transportation.
        // -----
        // Four wheels and an engine.
        // Carries seven people.
        // -----

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("-----");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("-----");
            }
        }
    }
}

```

```

}

// Output:
// TestCars2
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----


public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("-----");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
// Output:
// TestCars3
// -----
// A roof that opens up.
// Carries seven people.


public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("-----");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
// Output:
// TestCars4
// -----
// Standard transportation.
// Carries seven people.
}

// Define the base class, Car. The class defines two virtual methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is used, the base class method or the derived class method.
class Car
{
    public virtual void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
}

```

```
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}

}
```

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Versioning with the Override and New Keywords](#)
- [base](#)
- [abstract](#)

How to override the `ToString` method (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Every class or struct in C# implicitly inherits the `Object` class. Therefore, every object in C# gets the `ToString` method, which returns a string representation of that object. For example, all variables of type `int` have a `ToString` method, which enables them to return their contents as a string:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

When you create a custom class or struct, you should override the `ToString` method in order to provide information about your type to client code.

For information about how to use format strings and other types of custom formatting with the `ToString` method, see [Formatting Types](#).

IMPORTANT

When you decide what information to provide through this method, consider whether your class or struct will ever be used by untrusted code. Be careful to ensure that you do not provide any information that could be exploited by malicious code.

To override the `ToString` method in your class or struct:

1. Declare a `ToString` method with the following modifiers and return type:

```
public override string ToString(){}
```

2. Implement the method so that it returns a string.

The following example returns the name of the class in addition to the data specific to a particular instance of the class.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

You can test the `ToString` method as shown in the following code example:

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

See also

- [IFormattable](#)
- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Strings](#)
- [string](#)
- [override](#)
- [virtual](#)
- [Formatting Types](#)

Members (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Classes and structs have members that represent their data and behavior. A class's members include all the members declared in the class, along with all members (except constructors and finalizers) declared in all classes in its inheritance hierarchy. Private members in base classes are inherited but are not accessible from derived classes.

The following table lists the kinds of members a class or struct may contain:

MEMBER	DESCRIPTION
Fields	Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class. For example, a calendar class may have a field that contains the current date.
Constants	Constants are fields whose value is set at compile time and cannot be changed.
Properties	Properties are methods on a class that are accessed as if they were fields on that class. A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
Methods	Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter.
Events	Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates.
Operators	Overloaded operators are considered type members. When you overload an operator, you define it as a public static method in a type. For more information, see Operator overloading .
Indexers	Indexers enable an object to be indexed in a manner similar to arrays.
Constructors	Constructors are methods that are called when the object is first created. They are often used to initialize the data of an object.
Finalizers	Finalizers are used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately.

MEMBER	DESCRIPTION
Nested Types	Nested types are types declared within another type. Nested types are often used to describe objects that are used only by the types that contain them.

See also

- [C# Programming Guide](#)
- [Classes](#)

Abstract and Sealed Classes and Class Members (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `abstract` keyword enables you to create classes and `class` members that are incomplete and must be implemented in a derived class.

The `sealed` keyword enables you to prevent the inheritance of a class or certain class members that were previously marked `virtual`.

Abstract Classes and Class Members

Classes can be declared as abstract by putting the keyword `abstract` before the class definition. For example:

```
public abstract class A
{
    // Class members here.
}
```

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword `abstract` before the return type of the method. For example:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

If a `virtual` method is declared `abstract`, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, `DoWork` on class F cannot call `DoWork` on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Class Members

Classes can be declared as `sealed` by putting the keyword `sealed` before the class definition. For example:

```
public sealed class D
{
    // Class members here.
}
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the `sealed` keyword before the `override` keyword in the class member declaration. For example:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Inheritance](#)
- [Methods](#)

- Fields
- How to define abstract properties

Static Classes and Static Class Members (C# Programming Guide)

3/6/2021 • 5 minutes to read • [Edit Online](#)

A **static** class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the **new** operator to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself. For example, if you have a static class that is named `UtilityClass` that has a public static method named `MethodA`, you call the method as shown in the following example:

```
UtilityClass.MethodA();
```

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields. For example, in the .NET Class Library, the static `System.Math` class contains methods that perform mathematical operations, without any requirement to store or retrieve data that is unique to a particular instance of the `Math` class. That is, you apply the members of the class by specifying the class name and the method name, as shown in the following example.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

As is the case with all class types, the type information for a static class is loaded by the .NET runtime when the program that references the class is loaded. The program cannot specify exactly when the class is loaded. However, it is guaranteed to be loaded and to have its fields initialized and its static constructor called before the class is referenced for the first time in your program. A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

NOTE

To create a non-static class that allows only one instance of itself to be created, see [Implementing Singleton in C#](#).

The following list provides the main features of a static class:

- Contains only static members.
- Cannot be instantiated.
- Is sealed.
- Cannot contain [Instance Constructors](#).

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added. The

compiler will guarantee that instances of this class cannot be created.

Static classes are sealed and therefore cannot be inherited. They cannot inherit from any class except [Object](#). Static classes cannot contain an instance constructor. However, they can contain a static constructor. Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization. For more information, see [Static Constructors](#).

Example

Here is an example of a static class that contains two methods that convert temperature from Celsius to Fahrenheit and from Fahrenheit to Celsius:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
        }
    }
}
```

```

        break;
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Example Output:
Please select the convertor direction
1. From Celsius to Fahrenheit.
2. From Fahrenheit to Celsius.
:2
Please enter the Fahrenheit temperature: 20
Temperature in Celsius: -6.67
Press any key to exit.
*/

```

Static Members

A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it's explicitly passed in a method parameter.

It is more typical to declare a non-static class with some static members, than to declare an entire class as static. Two common uses of static fields are to keep a count of the number of objects that have been instantiated, or to store a value that must be shared among all instances.

Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.

Although a field cannot be declared as `static const`, a `const` field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, `const` fields can be accessed by using the same `ClassName.MemberName` notation that's used for static fields. No object instance is required.

C# does not support static local variables (that is, variables that are declared in method scope).

You declare static class members by using the `static` keyword before the return type of the member, as shown in the following example:

```

public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}

```

Static members are initialized before the static member is accessed for the first time and before the static constructor, if there is one, is called. To access a static class member, use the name of the class instead of a variable name to specify the location of the member, as shown in the following example:

```
Automobile.Drive();  
int i = Automobile.NumberOfWheels;
```

If your class contains static fields, provide a static constructor that initializes them when the class is loaded.

A call to a static method generates a call instruction in Microsoft intermediate language (MSIL), whereas a call to an instance method generates a `callvirt` instruction, which also checks for null object references. However, most of the time the performance difference between the two is not significant.

C# Language Specification

For more information, see [Static classes](#) and [Static and instance members](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [static](#)
- [Classes](#)
- [class](#)
- [Static Constructors](#)
- [Instance Constructors](#)

Access Modifiers (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

All types and type members have an accessibility level. The accessibility level controls whether they can be used from other code in your assembly or other assemblies. Use the following access modifiers to specify the accessibility of a type or member when you declare it:

- **public**: The type or member can be accessed by any other code in the same assembly or another assembly that references it.
- **private**: The type or member can be accessed only by code in the same `class` or `struct`.
- **protected**: The type or member can be accessed only by code in the same `class`, or in a `class` that is derived from that `class`.
- **internal**: The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal**: The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived `class` in another assembly.
- **private protected**: The type or member can be accessed only within its declaring assembly, by code in the same `class` or in a type that is derived from that `class`.

The following examples demonstrate how to specify access modifiers on a type and member:

```
public class Bicycle
{
    public void Pedal() { }
}
```

Not all access modifiers are valid for all types or members in all contexts. In some cases, the accessibility of a type member is constrained by the accessibility of its containing type.

Class, record, and struct accessibility

Classes, records, and structs declared directly within a namespace (in other words, that aren't nested within other classes or structs) can be either `public` or `internal`. `internal` is the default if no access modifier is specified.

Struct members, including nested classes and structs, can be declared `public`, `internal`, or `private`. Class members, including nested classes and structs, can be `public`, `protected internal`, `protected`, `internal`, `private protected`, or `private`. Class and struct members, including nested classes and structs, have `private` access by default. Private nested types aren't accessible from outside the containing type.

Derived classes and derived records can't have greater accessibility than their base types. You can't declare a public class `B` that derives from an internal class `A`. If allowed, it would have the effect of making `A` public, because all `protected` or `internal` members of `A` are accessible from the derived class.

You can enable specific other assemblies to access your internal types by using the `InternalsVisibleToAttribute`. For more information, see [Friend Assemblies](#).

Class, record, and struct member accessibility

Class and record members (including nested classes, records and structs) can be declared with any of the six

types of access. Struct members can't be declared as `protected`, `protected internal`, or `private protected` because structs don't support inheritance.

Normally, the accessibility of a member isn't greater than the accessibility of the type that contains it. However, a `public` member of an internal class might be accessible from outside the assembly if the member implements interface methods or overrides virtual methods that are defined in a public base class.

The type of any member field, property, or event must be at least as accessible as the member itself. Similarly, the return type and the parameter types of any method, indexer, or delegate must be at least as accessible as the member itself. For example, you can't have a `public` method `M` that returns a class `C` unless `C` is also `public`. Likewise, you can't have a `protected` property of type `A` if `A` is declared as `private`.

User-defined operators must always be declared as `public` and `static`. For more information, see [Operator overloading](#).

Finalizers can't have accessibility modifiers.

To set the access level for a `class`, `record`, or `struct` member, add the appropriate keyword to the member declaration, as shown in the following example.

```
// public class:  
public class Tricycle  
{  
    // protected method:  
    protected void Pedal() {}  
  
    // private field:  
    private int wheels = 3;  
  
    // protected internal property:  
    protected internal int Wheels  
    {  
        get { return wheels; }  
    }  
}
```

Other types

Interfaces declared directly within a namespace can be `public` or `internal` and, just like classes and structs, interfaces default to `internal` access. Interface members are `public` by default because the purpose of an interface is to enable other types to access a class or struct. Interface member declarations may include any access modifier. This is most useful for static methods to provide common implementations needed by all implementors of a class.

Enumeration members are always `public`, and no access modifiers can be applied.

Delegates behave like classes and structs. By default, they have `internal` access when declared directly within a namespace, and `private` access when nested.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)

- Interfaces
- private
- public
- internal
- protected
- protected internal
- private protected
- class
- struct
- interface

Fields (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

A *field* is a variable of any type that is declared directly in a [class](#) or [struct](#). Fields are *members* of their containing type.

A class or struct may have instance fields, static fields, or both. Instance fields are specific to an instance of a type. If you have a class T, with an instance field F, you can create two objects of type T, and modify the value of F in each object without affecting the value in the other object. By contrast, a static field belongs to the class itself, and is shared among all instances of that class. You can access the static field only by using the class name. If you access the static field by an instance name, you get [CS0176](#) compile-time error.

Generally, you should use fields only for variables that have private or protected accessibility. Data that your class exposes to client code should be provided through [methods](#), [properties](#), and [indexers](#). By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a *backing store* or *backing field*.

Fields typically store the data that must be accessible to more than one class method and must be stored for longer than the lifetime of any single method. For example, a class that represents a calendar date might have three integer fields: one for the month, one for the day, and one for the year. Variables that are not used outside the scope of a single method should be declared as *local variables* within the method body itself.

Fields are declared in the class block by specifying the access level of the field, followed by the type of the field, followed by the name of the field. For example:

```

public class CalendarEntry
{
    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
    }

    // public field (Generally not recommended).
    public string Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

To access a field in an object, add a period after the object name, followed by the name of the field, as in `objectname._fieldName`. For example:

```
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

A field can be given an initial value by using the assignment operator when the field is declared. To automatically assign the `Day` field to `"Monday"`, for example, you would declare `Day` as in the following example:

```
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Fields are initialized immediately before the constructor for the object instance is called. If the constructor assigns the value of a field, it will overwrite any value given during field declaration. For more information, see [Using Constructors](#).

NOTE

A field initializer cannot refer to other instance fields.

Fields can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#), or [private protected](#). These access modifiers define how users of the class can access the fields. For more information, see [Access Modifiers](#).

A field can optionally be declared [static](#). This makes the field available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members](#).

A field can be declared [readonly](#). A read-only field can only be assigned a value during initialization or in a constructor. A `static readonly` field is very similar to a constant, except that the C# compiler does not have access to the value of a static read-only field at compile time, only at run time. For more information, see [Constants](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Using Constructors](#)
- [Inheritance](#)
- [Access Modifiers](#)
- [Abstract and Sealed Classes and Class Members](#)

Constants (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the `const` modifier. Only the C# [built-in types](#) (excluding `System.Object`) may be declared as `const`. User-defined types, including classes, structs, and arrays, cannot be `const`. Use the `readonly` modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

C# does not support `const` methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example `int`, `uint`, `long`, and so on). For more information, see [enum](#).

Constants must be initialized as they are declared. For example:

```
class Calendar1
{
    public const int Months = 12;
}
```

In this example, the constant `Months` is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, `Months`), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, `const` fields cannot be passed by reference and cannot appear as an l-value in an expression.

NOTE

Use caution when you refer to constant values defined in other code such as DLLs. If a new version of the DLL defines a new value for the constant, your program will still hold the old literal value until it is recompiled against the new version.

Multiple constants of the same type can be declared at the same time, for example:

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

The expression that is used to initialize a constant can refer to another constant if it does not create a circular reference. For example:

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Constants can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) or [private protected](#). These access modifiers define how users of the class can access the constant. For more information, see [Access Modifiers](#).

Constants are accessed as if they were [static](#) fields because the value of the constant is the same for all instances of the type. You do not use the `static` keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example:

```
int birthstones = Calendar.Months;
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Properties](#)
- [Types](#)
- [readonly](#)
- [Immutability in C# Part One: Kinds of Immutability](#)

How to define abstract properties (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The following example shows how to define **abstract** properties. An abstract property declaration does not provide an implementation of the property accessors -- it declares that the class supports properties, but leaves the accessor implementation to derived classes. The following example demonstrates how to implement the abstract properties inherited from a base class.

This sample consists of three files, each of which is compiled individually and its resulting assembly is referenced by the next compilation:

- `abstractshape.cs`: the `Shape` class that contains an abstract `Area` property.
- `shapes.cs`: The subclasses of the `Shape` class.
- `shapetest.cs`: A test program to display the areas of some `Shape`-derived objects.

To compile the example, use the following command:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

This will create the executable file `shapetest.exe`.

Example

This file declares the `Shape` class that contains the `Area` property of the type `double`.

```

// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}

```

- Modifiers on the property are placed on the property declaration itself. For example:

```
public abstract double Area
```

- When declaring an abstract property (such as `Area` in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a `get` accessor is available, so the property is read-only.

Example

The following code shows three subclasses of `Shape` and how they override the `Area` property to provide their own implementation.

```

// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}

```

Example

The following code shows a test program that creates a number of `Shape`-derived objects and prints out their areas.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
/* Output:
   Shapes Collection
   Square #1 Area = 25.00
   Circle #1 Area = 28.27
   Rectangle #1 Area = 20.00
*/
```

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Properties](#)

How to define constants in C#

3/27/2021 • 2 minutes to read • [Edit Online](#)

Constants are fields whose values are set at compile time and can never be changed. Use constants to provide meaningful names instead of numeric literals ("magic numbers") for special values.

NOTE

In C# the `#define` preprocessor directive cannot be used to define constants in the way that is typically used in C and C++.

To define constant values of integral types (`int`, `byte`, and so on) use an enumerated type. For more information, see [enum](#).

To define non-integral constants, one approach is to group them in a single static class named `Constants`. This will require that all references to the constants be prefaced with the class name, as shown in the following example.

Example

```
using System;

static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

The use of the class name qualifier helps ensure that you and others who use the constant understand that it is constant and cannot be modified.

See also

- [Classes and Structs](#)

Properties (C# Programming Guide)

3/13/2021 • 4 minutes to read • [Edit Online](#)

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A `get` property accessor is used to return the property value, and a `set` property accessor is used to assign a new value. In C# 9 and later, an `init` property accessor is used to assign a new value only during object construction. These accessors can have different access levels. For more information, see [Restricting Accessor Accessibility](#).
- The `value` keyword is used to define the value being assigned by the `set` or `init` accessor.
- Properties can be *read-write* (they have both a `get` and a `set` accessor), *read-only* (they have a `get` accessor but no `set` accessor), or *write-only* (they have a `set` accessor, but no `get` accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as [auto-implemented properties](#).

Properties with backing fields

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The `get` accessor returns the value of the private field, and the `set` accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

The following example illustrates this pattern. In this example, the `TimePeriod` class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named `_seconds`. A read-write property named `Hours` allows the customer to specify the time interval in hours. Both the `get` and the `set` accessors perform the necessary conversion between hours and seconds. In addition, the `set` accessor validates the data and throws an [ArgumentOutOfRangeException](#) if the number of hours is invalid.

```

using System;

class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                    $"{nameof(value)} must be between 0 and 24.");
            _seconds = value * 3600;
        }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();
        // The property assignment causes the 'set' accessor to be called.
        t.Hours = 24;

        // Retrieving the property causes the 'get' accessor to be called.
        Console.WriteLine($"Time in hours: {t.Hours}");
    }
}
// The example displays the following output:
//     Time in hours: 24

```

Expression body definitions

Property accessors often consist of single-line statements that just assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the `=>` symbol followed by the expression to assign to or retrieve from the property.

Starting with C# 6, read-only properties can implement the `get` accessor as an expression-bodied member. In this case, neither the `get` accessor keyword nor the `return` keyword is used. The following example implements the read-only `Name` property as an expression-bodied member.

```
using System;

public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
    public static void Main()
    {
        var person = new Person("Magnus", "Hedlund");
        Console.WriteLine(person.Name);
    }
}

// The example displays the following output:
//      Magnus Hedlund
```

Starting with C# 7.0, both the `get` and the `set` accessor can be implemented as expression-bodied members. In this case, the `get` and `set` keywords must be present. The following example illustrates the use of expression body definitions for both accessors. Note that the `return` keyword is not used with the `get` accessor.

```

using System;

public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem("Shoes", 19.95m);
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95

```

Auto-implemented properties

In some cases, property `get` and `set` accessors just assign a value to or retrieve a value from a backing field without including any additional logic. By using auto-implemented properties, you can simplify your code while having the C# compiler transparently provide the backing field for you.

If a property has both a `get` and a `set` (or a `get` and an `init`) accessor, both must be auto-implemented. You define an auto-implemented property by using the `get` and `set` keywords without providing any implementation. The following example repeats the previous one, except that `Name` and `Price` are auto-implemented properties. The example also removes the parameterized constructor, so that `SaleItem` objects are now initialized with a call to the parameterless constructor and an [object initializer](#).

```
using System;

public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95
```

Related sections

- [Using Properties](#)
- [Interface Properties](#)
- [Comparison Between Properties and Indexers](#)
- [Restricting Accessor Accessibility](#)
- [Auto-Implemented Properties](#)

C# Language Specification

For more information, see [Properties](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Using Properties](#)
- [Indexers](#)
- [get keyword](#)
- [set keyword](#)

Using Properties (C# Programming Guide)

3/13/2021 • 8 minutes to read • [Edit Online](#)

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a `get` accessor and/or a `set` accessor. The code block for the `get` accessor is executed when the property is read; the code block for the `set` accessor is executed when the property is assigned a new value. A property without a `set` accessor is considered read-only. A property without a `get` accessor is considered write-only. A property that has both accessors is read-write. In C# 9 and later, you can use an `init` accessor instead of a `set` accessor to make the property read-only.

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a `ref` or `out` parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a `get`-accessor and/or a `set` accessor. For example:

```
public class Date
{
    private int _month = 7; // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

In this example, `Month` is declared as a property so that the `set` accessor can make sure that the `Month` value is set between 1 and 12. The `Month` property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see [Access Modifiers](#).

Auto-implemented properties provide simplified syntax for simple property declarations. For more information, see [Auto-Implemented Properties](#).

The `get` accessor

The body of the `get` accessor resembles that of a method. It must return a value of the property type. The execution of the `get` accessor is equivalent to reading the value of the field. For example, when you are returning the private variable from the `get` accessor and optimizations are enabled, the call to the `get`

accessor method is inlined by the compiler so there is no method-call overhead. However, a virtual `get` accessor method cannot be inlined because the compiler does not know at compile-time which method may actually be called at run time. The following is a `get` accessor that returns the value of a private field `_name`:

```
class Person
{
    private string _name; // the name field
    public string Name => _name; // the Name property
}
```

When you reference the property, except as the target of an assignment, the `get` accessor is invoked to read the value of the property. For example:

```
Person person = new Person();
//...

System.Console.WriteLine(person.Name); // the get accessor is invoked here
```

The `get` accessor must end in a `return` or `throw` statement, and control cannot flow off the accessor body.

It is a bad programming style to change the state of the object by using the `get` accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the `_number` field is accessed.

```
private int _number;
public int Number => _number++; // Don't do this
```

The `get` accessor can be used to return the field value or to compute it and return it. For example:

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

In the previous code segment, if you do not assign a value to the `Name` property, it will return the value `NA`.

The `set` accessor

The `set` accessor resembles a method whose return type is `void`. It uses an implicit parameter called `value`, whose type is the type of the property. In the following example, a `set` accessor is added to the `Name` property:

```
class Person
{
    private string _name; // the name field
    public string Name // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

When you assign a value to the property, the `set` accessor is invoked by using an argument that provides the new value. For example:

```
Person person = new Person();
person.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(person.Name); // the get accessor is invoked here
```

It is an error to use the implicit parameter name, `value`, for a local variable declaration in a `set` accessor.

The init accessor

The code to create an `init` accessor is the same as the code to create a `set` accessor except that you use the `init` keyword instead of `set`. The difference is that the `init` accessor can only be used in the constructor or by using an [object-initializer](#).

Remarks

Properties can be marked as `public`, `private`, `protected`, `internal`, `protected internal`, or `private protected`. These access modifiers define how users of the class can access the property. The `get` and `set` accessors for the same property may have different access modifiers. For example, the `get` may be `public` to allow read-only access from outside the type, and the `set` may be `private` or `protected`. For more information, see [Access Modifiers](#).

A property may be declared as a static property by using the `static` keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members](#).

A property may be marked as a virtual property by using the `virtual` keyword. This enables derived classes to override the property behavior by using the `override` keyword. For more information about these options, see [Inheritance](#).

A property overriding a virtual property can also be `sealed`, specifying that for derived classes it is no longer virtual. Lastly, a property can be declared `abstract`. This means that there is no implementation in the class, and derived classes must write their own implementation. For more information about these options, see [Abstract and Sealed Classes and Class Members](#).

NOTE

It is an error to use a `virtual`, `abstract`, or `override` modifier on an accessor of a `static` property.

Examples

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `NumberOfEmployees` by 1, and displays the Employee name and number.

```

public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's number:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
   Employee number: 108
   Employee name: Claude Vige
*/

```

Hidden property example

This example demonstrates how to access a property in a base class that is hidden by another property that has the same name in a derived class:

```

public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
   Name in the derived class is: John, Manager
   Name in the base class is: Mary
*/

```

The following are important points in the previous example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the `new` modifier is used in the declaration of the property in the derived class:

```
public new string Name
```

- The cast `(Employee)` is used to access the hidden property in the base class:

```
((Employee)m1).Name = "Mary";
```

For more information about hiding members, see the [new Modifier](#).

Override property example

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the `override` modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the

corresponding side for the square and cube.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.WriteLine("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.WriteLine("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
```

```
}

/* Example Output:
Enter the side: 4
Area of the square = 16.00
Area of the cube = 96.00

Enter the area: 24
Side of the square = 4.90
Side of the cube = 2.00
*/
```

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Interface Properties](#)
- [Auto-Implemented Properties](#)

Interface Properties (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Properties can be declared on an [interface](#). The following example declares an interface property accessor:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Interface properties typically don't have a body. The accessors indicate whether the property is read-write, read-only, or write-only. Unlike in classes and structs, declaring the accessors without a body doesn't declare an [auto-implemented property](#). Beginning with C# 8.0, an interface may define a default implementation for members, including properties. Defining a default implementation for a property in an interface is rare because interfaces may not define instance data fields.

Example

In this example, the interface `IEmployee` has a read-write property, `Name`, and a read-only property, `Counter`. The class `Employee` implements the `IEmployee` interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

You could use the fully qualified name of the property, which references the interface in which the member is declared. For example:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

The preceding example demonstrates [Explicit Interface Implementation](#). For example, if the class `Employee` is implementing two interfaces `ICitizen` and `IEmployee` and both interfaces have the `Name` property, the explicit interface member implementation will be necessary. That is, the following property declaration:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implements the `Name` property on the `IEmployee` interface, while the following declaration:

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implements the `Name` property on the `ICitizen` interface.

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

210 Hazem Abolrous

Sample output

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Using Properties](#)
- [Comparison Between Properties and Indexers](#)
- [Indexers](#)
- [Interfaces](#)

Restricting Accessor Accessibility (C# Programming Guide)

11/2/2020 • 4 minutes to read • [Edit Online](#)

The `get` and `set` portions of a property or indexer are called *accessors*. By default these accessors have the same visibility or access level of the property or indexer to which they belong. For more information, see [accessibility levels](#). However, it is sometimes useful to restrict access to one of these accessors. Typically, this involves restricting the accessibility of the `set` accessor, while keeping the `get` accessor publicly accessible. For example:

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

In this example, a property called `Name` defines a `get` and `set` accessor. The `get` accessor receives the accessibility level of the property itself, `public` in this case, while the `set` accessor is explicitly restricted by applying the `protected` access modifier to the accessor itself.

Restrictions on Access Modifiers on Accessors

Using the accessor modifiers on properties or indexers is subject to these conditions:

- You cannot use accessor modifiers on an interface or an explicit [interface](#) member implementation.
- You can use accessor modifiers only if the property or indexer has both `set` and `get` accessors. In this case, the modifier is permitted on only one of the two accessors.
- If the property or indexer has an `override` modifier, the accessor modifier must match the accessor of the overridden accessor, if any.
- The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.

Access Modifiers on Overriding Accessors

When you override a property or indexer, the overridden accessors must be accessible to the overriding code. Also, the accessibility of both the property/indexer and its accessors must match the corresponding overridden property/indexer and its accessors. For example:

```

public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}

public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}

```

Implementing Interfaces

When you use an accessor to implement an interface, the accessor may not have an access modifier. However, if you implement the interface using one accessor, such as `get`, the other accessor can have an access modifier, as in the following example:

```

public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}

```

Accessor Accessibility Domain

If you use an access modifier on the accessor, the [accessibility domain](#) of the accessor is determined by this modifier.

If you did not use an access modifier on the accessor, the accessibility domain of the accessor is determined by the accessibility level of the property or indexer.

Example

The following example contains three classes, `BaseClass`, `DerivedClass`, and `MainClass`. There are two properties on the `BaseClass`, `Name` and `Id` on both classes. The example demonstrates how the property `Id` on `DerivedClass` can be hidden by the property `Id` on `BaseClass` when you use a restrictive access modifier such as `protected` or `private`. Therefore, when you assign values to this property, the property on the `BaseClass` class is called instead. Replacing the access modifier by `public` will make the property accessible.

The example also demonstrates that a restrictive access modifier, such as `private` or `protected`, on the `set` accessor of the `Name` property in `DerivedClass` prevents access to the accessor and generates an error when you assign to it.

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }
    }

    // Using "protected" would make the set accessor not accessible.
    set
    {
        _name = value;
    }
}

// Using private on the following property hides it in the Main Class.
// Any assignment to the property will use Id in BaseClass.
new private string Id
{
    get
    {
        return _id;
    }
    set
    {
        _id = value;
    }
}

class MainClass
```

```

{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123"; // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Base: Name-BaseClass, ID-BaseClass
   Derived: John, ID-BaseClass
*/

```

Comments

Notice that if you replace the declaration `new private string Id` by `new public string Id`, you get the output:

`Name and ID in the base class: Name-BaseClass, ID-BaseClass`

`Name and ID in the derived class: John, John123`

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Indexers](#)
- [Access Modifiers](#)

How to declare and use read/write properties (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. This is accomplished through *accessors*: special methods that assign and retrieve values from the underlying data member. The `set` accessor enables data members to be assigned, and the `get` accessor retrieves data member values.

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties provide `get` and `set` accessors, so they are considered read/write properties.

Example

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }
        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();
```

```

// Print out the name and the age associated with the person:
Console.WriteLine("Person details - {0}", person);

// Set some values on the person object:
person.Name = "Joe";
person.Age = 99;
Console.WriteLine("Person details - {0}", person);

// Increment the Age property:
person.Age += 1;
Console.WriteLine("Person details - {0}", person);

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/*
 * Output:
 * Person details - Name = N/A, Age = 0
 * Person details - Name = Joe, Age = 99
 * Person details - Name = Joe, Age = 100
 */

```

Robust Programming

In the previous example, the `Name` and `Age` properties are [public](#) and include both a `get` and a `set` accessor. This allows any object to read and write these properties. It is sometimes desirable, however, to exclude one of the accessors. Omitting the `set` accessor, for example, makes the property read-only:

```

public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}

```

Alternatively, you can expose one accessor publicly but make the other private or protected. For more information, see [Asymmetric Accessor Accessibility](#).

Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

```

person.Name = "Joe";
person.Age = 99;

```

Note that in a property `set` method a special `value` variable is available. This variable contains the value that the user specified, for example:

```

_name = value;

```

Notice the clean syntax for incrementing the `Age` property on a `Person` object:

```
person.Age += 1;
```

If separate `set` and `get` methods were used to model properties, the equivalent code might look like this:

```
person.SetAge(person.GetAge() + 1);
```

The `ToString` method is overridden in this example:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Notice that `ToString` is not explicitly used in the program. It is invoked by default by the `WriteLine` calls.

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Classes and Structs](#)

Auto-Implemented Properties (C# Programming Guide)

3/13/2021 • 2 minutes to read • [Edit Online](#)

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's `get` and `set` accessors. In C# 9 and later, `init` accessors can also be declared as auto-implemented properties.

Example

The following example shows a simple class that has some auto-implemented properties:

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

You can't declare auto-implemented properties in interfaces. Auto-implemented properties declare a private instance backing field, and interfaces may not declare instance fields. Declaring a property in an interface without defining a body declares a property with accessors that must be implemented by each type that implements that interface.

In C# 6 and later, you can initialize auto-implemented properties similarly to fields:

```
public string FirstName { get; set; } = "Jane";
```

The class that is shown in the previous example is mutable. Client code can change the values in objects after creation. In complex classes that contain significant behavior (methods) as well as data, it's often necessary to have public properties. However, for small classes or structs that just encapsulate a set of values (data) and have little or no behaviors, you should use one of the following options for making the objects immutable:

- Declare only a `get` accessor (immutable everywhere except the constructor).
- Declare a `get` accessor and an `init` accessor (immutable everywhere except during object construction).
- Declare the `set` accessor as `private` (immutable to consumers).

For more information, see [How to implement a lightweight class with auto-implemented properties](#).

See also

- [Properties](#)
- [Modifiers](#)

How to implement a lightweight class with auto-implemented properties (C# Programming Guide)

3/13/2021 • 3 minutes to read • [Edit Online](#)

This example shows how to create an immutable lightweight class that serves only to encapsulate a set of auto-implemented properties. Use this kind of construct instead of a struct when you must use reference type semantics.

You can make an immutable property in the following ways:

- Declare only the `get` accessor, which makes the property immutable everywhere except in the type's constructor.
- Declare an `init` accessor instead of a `set` accessor, which makes the property settable only in the constructor or by using an [object initializer](#).
- Declare the `set` accessor to be `private`. The property is settable within the type, but it is immutable to consumers.

When you declare a `private` `set` accessor, you cannot use an object initializer to initialize the property. You must use a constructor or a factory method.

The following example shows how a property with only `get` accessor differs than one with `get` and `private set`.

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress
}
```

Example

The following example shows two ways to implement an immutable class that has auto-implemented properties. Each way declares one of the properties with a `private` `set` and one of the properties with a `get` only. The first class uses a constructor only to initialize the properties, and the second class uses a static factory method that calls a constructor.

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
```

```

    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams", "Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
                             "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i], addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();
    }
}

```

```
// List elements cannot be modified by client code.  
// CS0272:  
// list2[0].Name = "Eugene Zabokritski";  
  
// Keep the console open in debug mode.  
Console.WriteLine("Press any key to exit.");  
Console.ReadKey();  
}  
}  
  
/* Output:  
 Terry Adams, 123 Main St.  
 Fadi Fakhouri, 345 Cypress Ave.  
 Hanying Feng, 678 1st Ave  
 Cesar Garcia, 12 108th St.  
 Debra Garcia, 89 E. 42nd St.  
 */
```

The compiler creates backing fields for each auto-implemented property. The fields are not accessible directly from source code.

See also

- [Properties](#)
- [struct](#)
- [Object and Collection Initializers](#)

Methods (C# Programming Guide)

3/20/2021 • 10 minutes to read • [Edit Online](#)

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method.

The `Main` method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started. In an application that uses [top-level statements](#), the `Main` method is generated by the compiler and contains all top-level statements.

NOTE

This article discusses named methods. For information about anonymous functions, see [Anonymous Functions](#).

Method signatures

Methods are declared in a `class`, `struct`, or `interface` by specifying the access level such as `public` or `private`, optional modifiers such as `abstract` or `sealed`, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

IMPORTANT

A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains four methods:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() /* Method statements here */

    // Only derived classes can call this.
    protected void AddGas(int gallons) /* Method statements here */

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) /* Method statements here */ return 1;

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Method access

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the `Motorcycle` class can therefore be called as in the following example:

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

Method parameters vs. arguments

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code doesn't have to be the same as the parameter named defined in the method. For example:

```

public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}

```

Passing by reference vs. passing by value

By default, when an instance of a [value type](#) is passed to a method, its copy is passed instead of the instance itself. Therefore, changes to the argument have no effect on the original instance in the calling method. To pass a value-type instance by reference, use the `ref` keyword. For more information, see [Passing Value-Type Parameters](#).

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the

method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

You create a reference type by using the `class` keyword, as the following example shows:

```
public class SampleRefType
{
    public int value;
}
```

Now, if you pass an object that is based on this type to a method, a reference to the object is passed. The following example passes an object of type `SampleRefType` to method `ModifyObject`:

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

The example does essentially the same thing as the previous example in that it passes an argument by value to a method. But, because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `value` field of the parameter, `obj`, also changes the `value` field of the argument, `rt`, in the `TestRefType` method. The `TestRefType` method displays 33 as the output.

For more information about how to pass reference types by reference and by value, see [Passing Reference-Type Parameters](#) and [Reference Types](#).

Return values

Methods can return a value to the caller. If the return type, the type listed before the method name, is not `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a value that matches the return type will return that value to the method caller.

The value can be returned to the caller by value or, starting with C# 7.0, [by reference](#). Values are returned to the caller by reference if the `ref` keyword is used in the method signature and it follows each `return` keyword. For example, the following method signature and return statement indicate that the method returns a variable names `estDistance` by reference to the caller.

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

The `return` keyword also stops the execution of the method. If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the `return` keyword to return a value. For example, these two methods use the `return` keyword to return integers:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

To use a value returned by reference from a method, you must declare a `ref local` variable if you intend to modify its value. For example, if the `Planet.GetEstimatedDistance` method returns a `Double` value by reference, you can define it as a `ref local` variable with code like the following:

```
ref int distance = plant
```

Returning a multi-dimensional array from a method, `M`, that modifies the array's contents is not necessary if the calling function passed the array into `M`. You may return the resulting array from `M` for good style or functional flow of values, but it is not necessary because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the method `M`, any changes to the array's contents are observable by any code that has a reference to the array, as shown in the following example:

```

static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}

```

For more information, see [return](#).

Async methods

By using the `async` feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the `async` modifier, you can use the `await` operator in the method. When control reaches an `await` expression in the `async` method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

NOTE

An `async` method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the `async` method, whichever occurs first.

An `async` method typically has a return type of `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An `async` method that returns `void` can't be awaited, and the caller of a `void`-returning method can't catch exceptions that the method throws. Starting with C# 7.0, an `async` method can have [any task-like return type](#).

In the following example, `DelayAsync` is an `async` method that has a return type of `Task<TResult>`. `DelayAsync` has a `return` statement that returns an integer. Therefore the method declaration of `DelayAsync` must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer as the following statement demonstrates: `int result = await delayTask`.

The `Main` method is an example of an `async` method that has a return type of `Task`. It goes to the `DoSomethingAsync` method, and because it is expressed with a single line, it can omit the `async` and `await` keywords. Because `DoSomethingAsync` is an `async` method, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `await DoSomethingAsync();`.

```

using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5

```

An `async` method can't declare any `ref` or `out` parameters, but it can call methods that have such parameters.

For more information about `async` methods, see [Asynchronous programming with `async` and `await`](#) and [Async return types](#).

Expression body definitions

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using

`=>`:

```

public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);

```

If the method returns `void` or is an `async` method, then the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the `yield return` statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.

You call an iterator from client code by using a `foreach` statement.

The return type of an iterator can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

For more information, see [Iterators](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Access Modifiers](#)
- [Static Classes and Static Class Members](#)
- [Inheritance](#)
- [Abstract and Sealed Classes and Class Members](#)
- [params](#)
- [return](#)
- [out](#)
- [ref](#)
- [Passing Parameters](#)

Local functions (C# Programming Guide)

3/6/2021 • 9 minutes to read • [Edit Online](#)

Starting with C# 7.0, C# supports *local functions*. Local functions are private methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:

- Methods, especially iterator methods and async methods
- Constructors
- Property accessors
- Event accessors
- Anonymous methods
- Lambda expressions
- Finalizers
- Other local functions

However, local functions can't be declared inside an expression-bodied member.

NOTE

In some cases, you can use a lambda expression to implement functionality also supported by a local function. For a comparison, see [Local functions vs. lambda expressions](#).

Local functions make the intent of your code clear. Anyone reading your code can see that the method is not callable except by the containing method. For team projects, they also make it impossible for another developer to mistakenly call the method directly from elsewhere in the class or struct.

Local function syntax

A local function is defined as a nested method inside a containing member. Its definition has the following syntax:

```
<modifiers> <return-type> <method-name> <parameter-list>
```

You can use the following modifiers with a local function:

- `async`
- `unsafe`
- `static` (in C# 8.0 and later). A static local function can't capture local variables or instance state.
- `extern` (in C# 9.0 and later). An external local function must be `static`.

All local variables that are defined in the containing member, including its method parameters, are accessible in a non-static local function.

Unlike a method definition, a local function definition cannot include the member access modifier. Because all local functions are private, including an access modifier, such as the `private` keyword, generates compiler error CS0106, "The modifier 'private' is not valid for this item."

The following example defines a local function named `AppendPathSeparator` that is private to a method named

```
GetText :
```

```
private static string GetText(string path, string filename)
{
    var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
    var text = reader.ReadToEnd();
    return text;

    string AppendPathSeparator(string filepath)
    {
        return filepath.EndsWith(@"/") ? filepath : filepath + @"\";
    }
}
```

Beginning with C# 9.0, you can apply attributes to a local function, its parameters and type parameters, as the following example shows:

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

The preceding example uses a [special attribute](#) to assist the compiler in static analysis in a nullable context.

Local functions and exceptions

One of the useful features of local functions is that they can allow exceptions to surface immediately. For method iterators, exceptions are surfaced only when the returned sequence is enumerated, and not when the iterator is retrieved. For async methods, any exceptions thrown in an async method are observed when the returned task is awaited.

The following example defines an `OddSequence` method that enumerates odd numbers in a specified range. Because it passes a number greater than 100 to the `OddSequence` enumerator method, the method throws an [ArgumentOutOfRangeException](#). As the output from the example shows, the exception surfaces only when you iterate the numbers, and not when you retrieve the enumerator.

```

using System;
using System.Collections.Generic;

public class IteratorWithoutLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110);
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs) // line 11
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        for (int i = start; i <= end; i++)
        {
            if (i % 2 == 1)
                yield return i;
        }
    }
}

// The example displays the output like this:
//
//     Retrieved enumerator...
//     Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
//     (Parameter 'end')
//     at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32 end)+MoveNext() in
//     IteratorWithoutLocal.cs:line 22
//     at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line 11

```

If you put iterator logic into a local function, argument validation exceptions are thrown when you retrieve the enumerator, as the following example shows:

```

using System;
using System.Collections.Generic;

public class IteratorWithLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110); // line 8
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs)
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        return GetOddSequenceEnumerator();
    }

    IEnumerable<int> GetOddSequenceEnumerator()
    {
        for (int i = start; i <= end; i++)
        {
            if (i % 2 == 1)
                yield return i;
        }
    }
}

// The example displays the output like this:
//
//      Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
//(Parameter 'end')
//      at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in IteratorWithLocal.cs:line 22
//      at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8

```

Local functions vs. lambda expressions

At first glance, local functions and [lambda expressions](#) are very similar. In many cases, the choice between using lambda expressions and local functions is a matter of style and personal preference. However, there are real differences in where you can use one or the other that you should be aware of.

Let's examine the differences between the local function and lambda expression implementations of the factorial algorithm. Here's the version using a local function:

```

public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}

```

This version uses lambda expressions:

```

public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}

```

Naming

Local functions are explicitly named like methods. Lambda expressions are anonymous methods and need to be assigned to variables of a `delegate` type, typically either `Action` or `Func` types. When you declare a local function, the process is like writing a normal method; you declare a return type and a function signature.

Function signatures and lambda expression types

Lambda expressions rely on the type of the `Action` / `Func` variable that they're assigned to determine the argument and return types. In local functions, since the syntax is much like writing a normal method, argument types and return type are already part of the function declaration.

Definite assignment

Lambda expressions are objects that are declared and assigned at runtime. In order for a lambda expression to be used, it needs to be definitely assigned: the `Action` / `Func` variable that it will be assigned to must be declared and the lambda expression assigned to it. Notice that `LambdaFactorial` must declare and initialize the lambda expression `nthFactorial` before defining it. Not doing so results in a compile time error for referencing `nthFactorial` before assigning it.

Local functions are defined at compile time. As they're not assigned to variables, they can be referenced from any code location **where it is in scope**; in our first example `LocalFunctionFactorial`, we could declare our local function either above or below the `return` statement and not trigger any compiler errors.

These differences mean that recursive algorithms are easier to create using local functions. You can declare and define a local function that calls itself. Lambda expressions must be declared, and assigned a default value before they can be re-assigned to a body that references the same lambda expression.

Implementation as a delegate

Lambda expressions are converted to delegates when they're declared. Local functions are more flexible in that they can be written like a traditional method *or* as a delegate. Local functions are only converted to delegates when *used* as a delegate.

If you declare a local function and only reference it by calling it like a method, it will not be converted to a delegate.

Variable capture

The rules of [definite assignment](#) also affect any variables that are captured by the local function or lambda expression. The compiler can perform static analysis that enables local functions to definitely assign captured variables in the enclosing scope. Consider this example:

```

int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}

```

The compiler can determine that `LocalFunction` definitely assigns `y` when called. Because `LocalFunction` is called before the `return` statement, `y` is definitely assigned at the `return` statement.

Note that when a local function captures variables in the enclosing scope, the local function is implemented as a delegate type.

Heap allocations

Depending on their use, local functions can avoid heap allocations that are always necessary for lambda expressions. If a local function is never converted to a delegate, and none of the variables captured by the local function are captured by other lambdas or local functions that are converted to delegates, the compiler can avoid heap allocations.

Consider this async example:

```

public async Task<string> PerformLongRunningWorkLambda(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    };

    return await longRunningWorkImplementation();
}

```

The closure for this lambda expression contains the `address`, `index` and `name` variables. In the case of local functions, the object that implements the closure may be a `struct` type. That struct type would be passed by reference to the local function. This difference in implementation would save on an allocation.

The instantiation necessary for lambda expressions means extra memory allocations, which may be a performance factor in time-critical code paths. Local functions do not incur this overhead. In the example above, the local functions version has two fewer allocations than the lambda expression version.

If you know that your local function won't be converted to a delegate and none of the variables captured by it are captured by other lambdas or local functions that are converted to delegates, you can guarantee that your local function avoids being allocated on the heap by declaring it as a `static` local function. Note that this feature is available in C# 8.0 and newer.

NOTE

The local function equivalent of this method also uses a class for the closure. Whether the closure for a local function is implemented as a `class` or a `struct` is an implementation detail. A local function may use a `struct` whereas a lambda will always use a `class`.

```
public async Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Usage of the `yield` keyword

One final advantage not demonstrated in this sample is that local functions can be implemented as iterators, using the `yield return` syntax to produce a sequence of values.

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

The `yield return` statement is not allowed in lambda expressions, see [compiler error CS1621](#).

While local functions may seem redundant to lambda expressions, they actually serve different purposes and have different uses. Local functions are more efficient for the case when you want to write a function that is called only from the context of another method.

See also

- [Methods](#)

Ref returns and ref locals

3/12/2020 • 6 minutes to read • [Edit Online](#)

Starting with C# 7.0, C# supports reference return values (ref returns). A reference return value allows a method to return a reference to a variable, rather than a value, back to a caller. The caller can then choose to treat the returned variable as if it were returned by value or by reference. The caller can create a new variable that is itself a reference to the returned value, called a ref local.

What is a reference return value?

Most developers are familiar with passing an argument to a called method *by reference*. A called method's argument list includes a variable passed by reference. Any changes made to its value by the called method are observed by the caller. A *reference return value* means that a method returns a *reference* (or an alias) to some variable. That variable's scope must include the method. That variable's lifetime must extend beyond the return of the method. Modifications to the method's return value by the caller are made to the variable that is returned by the method.

Declaring that a method returns a *reference return value* indicates that the method returns an alias to a variable. The design intent is often that the calling code should have access to that variable through the alias, including to modify it. It follows that methods returning by reference can't have the return type `void`.

There are some restrictions on the expression that a method can return as a reference return value. Restrictions include:

- The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method. Attempting to return a local variable generates compiler error CS8168, "Cannot return local 'obj' by reference because it is not a ref local."
- The return value cannot be the literal `null`. Returning `null` generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

A method with a ref return can return an alias to a variable whose value is currently the null (uninstantiated) value or a [nullable value type](#) for a value type.

- The return value cannot be a constant, an enumeration member, the by-value return value from a property, or a method of a `class` or `struct`. Violating this rule generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

In addition, reference return values are not allowed on async methods. An asynchronous method may return before it has finished execution, while its return value is still unknown.

Defining a ref return value

A method that returns a *reference return value* must satisfy the following two conditions:

- The method signature includes the `ref` keyword in front of the return type.
- Each `return` statement in the method body includes the `ref` keyword in front of the name of the returned instance.

The following example shows a method that satisfies those conditions and returns a reference to a `Person` object named `p`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Consuming a ref return value

The ref return value is an alias to another variable in the called method's scope. You can interpret any use of the ref return as using the variable it aliases:

- When you assign its value, you are assigning a value to the variable it aliases.
- When you read its value, you are reading the value of the variable it aliases.
- If you return it *by reference*, you are returning an alias to that same variable.
- If you pass it to another method *by reference*, you are passing a reference to the variable it aliases.
- When you make a **ref local** alias, you make a new alias to the same variable.

Ref locals

Assume the `GetContactInformation` method is declared as a ref return:

```
public ref Person GetContactInformation(string fname, string lname)
```

A by-value assignment reads the value of a variable and assigns it to a new variable:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

The preceding assignment declares `p` as a local variable. Its initial value is copied from reading the value returned by `GetContactInformation`. Any future assignments to `p` will not change the value of the variable returned by `GetContactInformation`. The variable `p` is no longer an alias to the variable returned.

You declare a **ref/local**/variable to copy the alias to the original value. In the following assignment, `p` is an alias to the variable returned from `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

Subsequent usage of `p` is the same as using the variable returned by `GetContactInformation` because `p` is an alias for that variable. Changes to `p` also change the variable returned from `GetContactInformation`.

The `ref` keyword is used both before the local variable declaration *and* before the method call.

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how one can define a ref local value that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

The `ref` keyword is used both before the local variable declaration *and* before the value in the second example. Failure to include both `ref` keywords in the variable declaration and assignment in both examples results in compiler error CS8172, "Cannot initialize a by-reference variable with a value."

Prior to C# 7.3, ref local variables couldn't be reassigned to refer to different storage after being initialized. That

restriction has been removed. The following example shows a reassignment:

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Ref local variables must still be initialized when they are declared.

Ref returns and ref locals: an example

The following example defines a `NumberStore` class that stores an array of integer values. The `FindNumber` method returns by reference the first number that is greater than or equal to the number passed as an argument. If no number is greater than or equal to the argument, the method returns the number in index 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

The following example calls the `NumberStore.FindNumber` method to retrieve the first value that is greater than or equal to 16. The caller then doubles the value returned by the method. The output from the example shows the change reflected in the value of the array elements of the `NumberStore` instance.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//      Original sequence: 1 3 7 15 31 63 127 255 511 1023
//      New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Without support for reference return values, such an operation is performed by returning the index of the array element along with its value. The caller can then use this index to modify the value in a separate method call. However, the caller can also modify the index to access and possibly modify other array values.

The following example shows how the `FindNumber` method could be rewritten after C# 7.3 to use ref local reassignment:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

This second version is more efficient with longer sequences in scenarios where the number sought is closer to the end of the array.

See also

- [ref keyword](#)
- [Write safe efficient code](#)

Passing Parameters (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

In C#, arguments can be passed to parameters either by value or by reference. Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment. To pass a parameter by reference with the intent of changing the value, use the `ref`, or `out` keyword. To pass by reference with the intent of avoiding copying but not changing the value, use the `in` modifier. For simplicity, only the `ref` keyword is used in the examples in this topic. For more information about the difference between `in`, `ref`, and `out`, see [in, ref, and out](#).

The following example illustrates the difference between value and reference parameters.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

For more information, see the following topics:

- [Passing Value-Type Parameters](#)
- [Passing Reference-Type Parameters](#)

C# Language Specification

For more information, see [Argument lists](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Methods](#)

Passing Value-Type Parameters (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

A [value-type](#) variable contains its data directly as opposed to a [reference-type](#) variable, which contains a reference to its data. Passing a value-type variable to a method by value means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no effect on the original data stored in the argument variable. If you want the called method to change the value of the argument, you must pass it by reference, using the [ref](#) or [out](#) keyword. You may also use the [in](#) keyword to pass a value parameter by reference to avoid the copy while guaranteeing that the value will not be changed. For simplicity, the following examples use [ref](#).

Passing Value Types by Value

The following example demonstrates passing value-type parameters by value. The variable [n](#) is passed by value to the method [SquareIt](#). Any changes that take place inside the method have no effect on the original value of the variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
        // The parameter x is passed by value.
        // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n); // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

The variable [n](#) is a value type. It contains its data, the value [5](#). When [SquareIt](#) is invoked, the contents of [n](#) are copied into the parameter [x](#), which is squared inside the method. In [Main](#), however, the value of [n](#) is the same after calling the [SquareIt](#) method as it was before. The change that takes place inside the method only affects the local variable [x](#).

Passing Value Types by Reference

The following example is the same as the previous example, except that the argument is passed as a `ref` parameter. The value of the underlying argument, `n`, is changed when `x` is changed in the method.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
        // The parameter x is passed by reference.
        // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n); // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

In this example, it is not the value of `n` that is passed; rather, a reference to `n` is passed. The parameter `x` is not an `int`; it is a reference to an `int`, in this case, a reference to `n`. Therefore, when `x` is squared inside the method, what actually is squared is what `x` refers to, `n`.

Swapping Value Types

A common example of changing the values of arguments is a swap method, where you pass two variables to the method, and the method swaps their contents. You must pass the arguments to the swap method by reference. Otherwise, you swap local copies of the parameters inside the method, and no change occurs in the calling method. The following example swaps integer values.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

When you call the `SwapByRef` method, use the `ref` keyword in the call, as shown in the following example.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

/* Output:
   i = 2  j = 3
   i = 3  j = 2
*/
```

See also

- [C# Programming Guide](#)
- [Passing Parameters](#)
- [Passing Reference-Type Parameters](#)

Passing Reference-Type Parameters (C# Programming Guide)

11/2/2020 • 4 minutes to read • [Edit Online](#)

A variable of a [reference type](#) does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data belonging to the referenced object, such as the value of a class member. However, you cannot change the value of the reference itself; for example, you cannot use the same reference to allocate memory for a new object and have it persist outside the method. To do that, pass the parameter using the [ref](#) or [out](#) keyword. For simplicity, the following examples use [ref](#).

Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, `arr`, by value, to a method, `Change`. Because the parameter is a reference to `arr`, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888; // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

In the preceding example, the array, `arr`, which is a reference type, is passed to the method without the `ref` parameter. In such a case, a copy of the reference, which points to `arr`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element, in this case from `1` to `888`. However, allocating a new portion of memory by using the `new` operator inside the `Change` method makes the variable `pArray` reference a new array. Thus, any changes after that will not affect the original array, `arr`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

Passing Reference Types by Reference

The following example is the same as the previous example, except that the `ref` keyword is added to the method header and call. Any changes that take place in the method affect the original variable in the calling program.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
        arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}",
        arr[0]);
    }
    /* Output:
     * Inside Main, before calling the method, the first element is: 1
     * Inside the method, the first element is: -3
     * Inside Main, after calling the method, the first element is: -3
     */
}
```

All of the changes that take place inside the method affect the original array in `Main`. In fact, the original array is reallocated using the `new` operator. Thus, after calling the `Change` method, any reference to `arr` points to the five-element array, which is created in the `Change` method.

Swapping Two Strings

Swapping strings is a good example of passing reference-type parameters by reference. In the example, two strings, `str1` and `str2`, are initialized in `Main` and passed to the `SwapStrings` method as parameters modified by the `ref` keyword. The two strings are swapped inside the method and inside `Main` as well.

```

class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
        // The string parameter is passed by reference.
        // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);    // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
   Inside Main, before swapping: John Smith
   Inside the method: Smith John
   Inside Main, after swapping: Smith John
*/

```

In this example, the parameters need to be passed by reference to affect the variables in the calling program. If you remove the `ref` keyword from both the method header and the method call, no changes will take place in the calling program.

For more information about strings, see [string](#).

See also

- [C# Programming Guide](#)
- [Passing Parameters](#)
- [ref](#)
- [in](#)
- [out](#)
- [Reference Types](#)

How to know the difference between passing a struct and passing a class reference to a method (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The following example demonstrates how passing a `struct` to a method differs from passing a `class` instance to a method. In the example, both of the arguments (struct and class instance) are passed by value, and both methods change the value of one field of the argument. However, the results of the two methods are not the same because what is passed when you pass a struct differs from what is passed when you pass an instance of a class.

Because a struct is a [value type](#), when you [pass a struct by value](#) to a method, the method receives and operates on a copy of the struct argument. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a [reference type](#), not a value type. When [a reference type is passed by value](#) to a method, the method receives a copy of the reference to the class instance. That is, the called method receives a copy of the address of the instance, and the calling method retains the original address of the instance. The class instance in the calling method has an address, the parameter in the called method has a copy of the address, and both addresses refer to the same object. Because the parameter contains only a copy of the address, the called method cannot change the address of the class instance in the calling method. However, the called method can use the copy of the address to access the class members that both the original address and the copy of the address reference. If the called method changes a class member, the original class instance in the calling method also changes.

The output of the following example illustrates the difference. The value of the `willIChange` field of the class instance is changed by the call to method `ClassTaker` because the method uses the address in the parameter to find the specified field of the class instance. The `willIChange` field of the struct in the calling method is not changed by the call to method `StructTaker` because the value of the argument is a copy of the struct itself, not a copy of its address. `StructTaker` changes the copy, and the copy is lost when the call to `StructTaker` is completed.

Example

```

using System;

class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

See also

- [C# Programming Guide](#)
- [Classes](#)
- [Structure types](#)
- [Passing Parameters](#)

Implicitly typed local variables (C# Programming Guide)

11/2/2020 • 5 minutes to read • [Edit Online](#)

Local variables can be declared without giving an explicit type. The `var` keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET class library. For more information about how to initialize arrays with `var`, see [Implicitly Typed Arrays](#).

The following examples show various ways in which local variables can be declared with `var`:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

It is important to understand that the `var` keyword does not mean "variant" and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The `var` keyword may be used in the following contexts:

- On local variables (variables declared at method scope) as shown in the previous example.
- In a `for` initialization statement.

```
for (var x = 1; x < 10; x++)
```

- In a `foreach` initialization statement.

```
foreach (var item in list) {...}
```

- In a `using` statement.

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

For more information, see [How to use implicitly typed local variables and arrays in a query expression](#).

var and anonymous types

In many cases the use of `var` is optional and is just a syntactic convenience. However, when a variable is initialized with an anonymous type you must declare the variable as `var` if you need to access the properties of the object at a later point. This is a common scenario in LINQ query expressions. For more information, see [Anonymous Types](#).

From the perspective of your source code, an anonymous type has no name. Therefore, if a query variable has been initialized with `var`, then the only way to access the properties in the returned sequence of objects is to use `var` as the type of the iteration variable in the `foreach` statement.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUEBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
            from w in words
            select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
    /* Outputs:
       Uppercase: APPLE, Lowercase: apple
       Uppercase: BLUEBERRY, Lowercase: blueberry
       Uppercase: CHERRY, Lowercase: cherry
    */
}
```

Remarks

The following restrictions apply to implicitly-typed variable declarations:

- `var` can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
- `var` cannot be used on fields at class scope.
- Variables declared by using `var` cannot be used in the initialization expression. In other words, this expression is legal: `int i = (i = 20);` but this expression produces a compile-time error:
`var i = (i = 20);`
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- If a type named `var` is in scope, then the `var` keyword will resolve to that type name and will not be treated as part of an implicitly typed local variable declaration.

Implicit typing with the `var` keyword can only be applied to variables at local method scope. Implicit typing is not available for class fields as the C# compiler would encounter a logical paradox as it processed the code: the compiler needs to know the type of the field, but it cannot determine the type until the assignment expression is

analyzed, and the expression cannot be evaluated without knowing the type. Consider the following code:

```
private var bookTitles;
```

`bookTitles` is a class field given the type `var`. As the field has no expression to evaluate, it is impossible for the compiler to infer what type `bookTitles` is supposed to be. In addition, adding an expression to the field (like you would for a local variable) is also insufficient:

```
private var bookTitles = new List<string>();
```

When the compiler encounters fields during code compilation, it records each field's type before processing any expressions associated with it. The compiler encounters the same paradox trying to parse `bookTitles`: it needs to know the type of the field, but the compiler would normally determine `var`'s type by analyzing the expression, which isn't possible without knowing the type beforehand.

You may find that `var` can also be useful with query expressions in which the exact constructed type of the query variable is difficult to determine. This can occur with grouping and ordering operations.

The `var` keyword can also be useful when the specific type of the variable is tedious to type on the keyboard, or is obvious, or does not add to the readability of the code. One example where `var` is helpful in this manner is with nested generic types such as those used with group operations. In the following query, the type of the query variable is `IEnumerable<IGrouping<string, Student>>`. As long as you and others who must maintain your code understand this, there is no problem with using implicit typing for convenience and brevity.

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an IEnumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

The use of `var` helps simplify your code, but its use should be restricted to cases where it is required, or when it makes your code easier to read. For more information about when to use `var` properly, see the [Implicitly typed local variables](#) section on the C# Coding Guidelines article.

See also

- [C# Reference](#)
- [Implicitly Typed Arrays](#)
- [How to use implicitly typed local variables and arrays in a query expression](#)
- [Anonymous Types](#)
- [Object and Collection Initializers](#)
- [var](#)
- [LINQ in C#](#)
- [LINQ \(Language-Integrated Query\)](#)
- [for](#)
- [foreach, in](#)
- [using Statement](#)

How to use implicitly typed local variables and arrays in a query expression (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

You can use implicitly typed local variables whenever you want the compiler to determine the type of a local variable. You must use implicitly typed local variables to store anonymous types, which are often used in query expressions. The following examples illustrate both optional and required uses of implicitly typed local variables in queries.

Implicitly typed local variables are declared by using the `var` contextual keyword. For more information, see [Implicitly Typed Local Variables](#) and [Implicitly Typed Arrays](#).

Example

The following example shows a common scenario in which the `var` keyword is required: a query expression that produces a sequence of anonymous types. In this scenario, both the query variable and the iteration variable in the `foreach` statement must be implicitly typed by using `var` because you do not have access to a type name for the anonymous type. For more information about anonymous types, see [Anonymous Types](#).

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

Example

The following example uses the `var` keyword in a situation that is similar, but in which the use of `var` is optional. Because `student.LastName` is a string, execution of the query returns a sequence of strings. Therefore, the type of `queryID` could be declared as `System.Collections.Generic.IEnumerable<string>` instead of `var`. Keyword `var` is used for convenience. In the example, the iteration variable in the `foreach` statement is explicitly typed as a string, but it could instead be declared by using `var`. Because the type of the iteration variable is not an anonymous type, the use of `var` is an option, not a requirement. Remember, `var` itself is not a type, but an instruction to the compiler to infer and assign the type.

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

See also

- [C# Programming Guide](#)
- [Extension Methods](#)
- [LINQ \(Language-Integrated Query\)](#)
- [var](#)
- [LINQ in C#](#)

Extension Methods (C# Programming Guide)

11/2/2020 • 9 minutes to read • [Edit Online](#)

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they're called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there's no apparent difference between calling an extension method and the methods defined in a type.

The most common extension methods are the LINQ standard query operators that add query functionality to the existing [System.Collections.IEnumerable](#) and [System.Collections.Generic.IEnumerable<T>](#) types. To use the standard query operators, first bring them into scope with a `using System.Linq` directive. Then any type that implements [IEnumerable<T>](#) appears to have instance methods such as [GroupBy](#), [OrderBy](#), [Average](#), and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an [IEnumerable<T>](#) type such as [List<T>](#) or [Array](#).

OrderBy Example

The following example shows how to call the standard query operator `OrderBy` method on an array of integers. The expression in parentheses is a lambda expression. Many standard query operators take lambda expressions as parameters, but this isn't a requirement for extension methods. For more information, see [Lambda Expressions](#).

```
class ExtensionMethods2
{
    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
} //Output: 10 15 21 26 39 45
```

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on. The parameter is preceded by the `this` modifier.

Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

The following example shows an extension method defined for the [System.String](#) class. It's defined inside a non-nested, non-generic static class:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                            StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

The `WordCount` extension method can be brought into scope with this `using` directive:

```
using ExtensionMethods;
```

And it can be called from an application by using this syntax:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

You invoke the extension method in your code with instance method syntax. The intermediate language (IL) generated by the compiler translates your code into a call on the static method. The principle of encapsulation is not really being violated. Extension methods cannot access private variables in the type they are extending.

For more information, see [How to implement and call a custom extension method](#).

In general, you'll probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code. To enable extension methods for a particular type, just add a `using` directive for the namespace in which the methods are defined. For example, to use the standard query operators, add this `using` directive to your code:

```
using System.Linq;
```

(You may also have to add a reference to `System.Core.dll`.) You'll notice that the standard query operators now appear in IntelliSense as additional methods available for most `IEnumerable<T>` types.

Binding Extension Methods at Compile Time

You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than instance methods defined in the type itself. In other words, if a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.

Example

The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class `Extensions` contains

extension methods defined for any type that implements `IMyInterface`. Classes `A`, `B`, and `C` all implement the interface.

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes.

When the compiler can't find an instance method with a matching signature, it will bind to a matching extension method if one exists.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }
}
```

```

        ,
class B : IMyInterface
{
    public void MethodB() { Console.WriteLine("B.MethodB()"); }
    public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
}

class C : IMyInterface
{
    public void MethodB() { Console.WriteLine("C.MethodB()"); }
    public void MethodA(object obj)
    {
        Console.WriteLine("C.MethodA(object obj)");
    }
}

class ExtMethodDemo
{
    static void Main(string[] args)
    {
        // Declare an instance of class A, class B, and class C.
        A a = new A();
        B b = new B();
        C c = new C();

        // For a, b, and c, call the following methods:
        //      -- MethodA with an int argument
        //      -- MethodA with a string argument
        //      -- MethodB with no argument.

        // A contains no MethodA, so each call to MethodA resolves to
        // the extension method that has a matching signature.
        a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
        a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

        // A has a method that matches the signature of the following call
        // to MethodB.
        a.MethodB();            // A.MethodB()

        // B has methods that match the signatures of the following
        // method calls.
        b.MethodA(1);           // B.MethodA(int)
        b.MethodB();             // B.MethodB()

        // B has no matching method for the following call, but
        // class Extension does.
        b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

        // C contains an instance method that matches each of the following
        // method calls.
        c.MethodA(1);           // C.MethodA(object)
        c.MethodA("hello");     // C.MethodA(object)
        c.MethodB();             // C.MethodB()
    }
}
/*
/* Output:
Extension.MethodA(this IMyInterface myInterface, int i)
Extension.MethodA(this IMyInterface myInterface, string s)
A.MethodB()
B.MethodA(int i)
B.MethodB()
Extension.MethodA(this IMyInterface myInterface, string s)
C.MethodA(object obj)
C.MethodA(object obj)
C.MethodB()
*/

```

Common Usage Patterns

Collection Functionality

In the past, it was common to create "Collection Classes" that implemented the `System.Collections.Generic.IEnumerable<T>` interface for a given type and contained functionality that acted on collections of that type. While there's nothing wrong with creating this type of collection object, the same functionality can be achieved by using an extension on the `System.Collections.Generic.IEnumerable<T>`. Extensions have the advantage of allowing the functionality to be called from any collection such as an `System.Array` or `System.Collections.Generic.List<T>` that implements `System.Collections.Generic.IEnumerable<T>` on that type. An example of this using an Array of Int32 can be found [earlier in this article](#).

Layer-Specific Functionality

When using an Onion Architecture or other layered application design, it's common to have a set of Domain Entities or Data Transfer Objects that can be used to communicate across application boundaries. These objects generally contain no functionality, or only minimal functionality that applies to all layers of the application. Extension methods can be used to add functionality that is specific to each application layer without loading the object down with methods not needed or wanted in other layers.

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Extending Predefined Types

Rather than creating new objects when reusable functionality needs to be created, we can often extend an existing type, such as a .NET or CLR type. As an example, if we don't use extension methods, we might create an `Engine` or `Query` class to do the work of executing a query on a SQL Server that may be called from multiple places in our code. However we can instead extend the `System.Data.SqlClient.SqlConnection` class using extension methods to perform that query from anywhere we have a connection to a SQL Server. Other examples might be to add common functionality to the `System.String` class, extend the data processing capabilities of the `System.IO.File` and `System.IO.Stream` objects, and `System.Exception` objects for specific error handling functionality. These types of use-cases are limited only by your imagination and good sense.

Extending predefined types can be difficult with `struct` types because they're passed by value to methods. That means any changes to the struct are made to a copy of the struct. Those changes aren't visible once the extension method exits. Beginning with C# 7.2, you can add the `ref` modifier to the first argument of an extension method. Adding the `ref` modifier means the first argument is passed by reference. This enables you to write extension methods that change the state of the struct being extended.

General Guidelines

While it's still considered preferable to add functionality by modifying an object's code or deriving a new type whenever it's reasonable and possible to do so, extension methods have become a crucial option for creating reusable functionality throughout the .NET ecosystem. For those occasions when the original source isn't under your control, when a derived object is inappropriate or impossible, or when the functionality shouldn't be

exposed beyond its applicable scope, Extension methods are an excellent choice.

For more information on derived types, see [Inheritance](#).

When using an extension method to extend a type whose source code you aren't in control of, you run the risk that a change in the implementation of the type will cause your extension method to break.

If you do implement extension methods for a given type, remember the following points:

- An extension method will never be called if it has the same signature as a method defined in the type.
- Extension methods are brought into scope at the namespace level. For example, if you have multiple static classes that contain extension methods in a single namespace named `Extensions`, they'll all be brought into scope by the `using Extensions;` directive.

For a class library that you implemented, you shouldn't use extension methods to avoid incrementing the version number of an assembly. If you want to add significant functionality to a library for which you own the source code, follow the .NET guidelines for assembly versioning. For more information, see [Assembly Versioning](#).

See also

- [C# Programming Guide](#)
- [Parallel Programming Samples](#) (these include many example extension methods)
- [Lambda Expressions](#)
- [Standard Query Operators Overview](#)
- [Conversion rules for Instance parameters and their impact](#)
- [Extension methods Interoperability between languages](#)
- [Extension methods and Curried Delegates](#)
- [Extension method Binding and Error reporting](#)

How to implement and call a custom extension method (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

This topic shows how to implement your own extension methods for any .NET type. Client code can use your extension methods by adding a reference to the DLL that contains them, and adding a `using` directive that specifies the namespace in which the extension methods are defined.

To define and call the extension method

1. Define a static `class` to contain the extension method.

The class must be visible to client code. For more information about accessibility rules, see [Access Modifiers](#).

2. Implement the extension method as a static method with at least the same visibility as the containing class.
3. The first parameter of the method specifies the type that the method operates on; it must be preceded with the `this` modifier.
4. In the calling code, add a `using` directive to specify the `namespace` that contains the extension method class.
5. Call the methods as if they were instance methods on the type.

Note that the first parameter is not specified by calling code because it represents the type on which the operator is being applied, and the compiler already knows the type of your object. You only have to provide arguments for parameters 2 through `n`.

Example

The following example implements an extension method named `WordCount` in the `CustomExtensions.StringExtension` class. The method operates on the `String` class, which is specified as the first method parameter. The `CustomExtensions` namespace is imported into the application namespace, and the method is called inside the `Main` method.

```

using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.', '?'}, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}

```

.NET Security

Extension methods present no specific security vulnerabilities. They can never be used to impersonate existing methods on a type, because all name collisions are resolved in favor of the instance or static method defined by the type itself. Extension methods cannot access any private data in the extended class.

See also

- [C# Programming Guide](#)
- [Extension Methods](#)
- [LINQ \(Language-Integrated Query\)](#)
- [Static Classes and Static Class Members](#)
- [protected](#)
- [internal](#)
- [public](#)
- [this](#)
- [namespace](#)

How to create a new method for an enumeration (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

You can use extension methods to add functionality specific to a particular enum type.

Example

In the following example, the `Grades` enumeration represents the possible letter grades that a student may receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
}

/* Output:
First is a passing grade.
Second is not a passing grade.

Raising the bar!

First is not a passing grade.
Second is not a passing grade.
*/
```

Note that the `Extensions` class also contains a static variable that is updated dynamically and that the return

value of the extension method reflects the current value of that variable. This demonstrates that, behind the scenes, extension methods are invoked directly on the static class in which they are defined.

See also

- [C# Programming Guide](#)
- [Extension Methods](#)

Named and Optional Arguments (C# Programming Guide)

3/6/2021 • 7 minutes to read • [Edit Online](#)

C# 4 introduces named and optional arguments. *Named arguments* enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list. *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters enable you to supply arguments for selected parameters. This capability greatly eases calls to COM interfaces such as the Microsoft Office Automation APIs.

Named Arguments

Named arguments free you from matching the order of parameters in the parameter lists of called methods. The parameter for each argument can be specified by parameter name. For example, a function that prints order details (such as, seller name, order number & product name) can be called by sending arguments by position, in the order defined by the function.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

If you don't remember the order of the parameters but know their names, you can send the arguments in any order.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Named arguments also improve the readability of your code by identifying what each argument represents. In the example method below, the `sellerName` can't be null or white space. As both `sellerName` and `productName` are string types, instead of sending arguments by position, it makes sense to use named arguments to disambiguate the two and reduce confusion for anyone reading the code.

Named arguments, when used with positional arguments, are valid as long as

- they're not followed by any positional arguments, or

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- *starting with C# 7.2*, they're used in the correct position. In the example below, the parameter `orderNum` is in the correct position but isn't explicitly named.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Positional arguments that follow any out-of-order named arguments are invalid.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Example

The following code implements the examples from this section along with some additional ones.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");      // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                  // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Optional Arguments

The definition of a method, constructor, indexer, or delegate can specify its parameters are required or optional. Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.

Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. A default value must be one of the following types of expressions:

- a constant expression;
- an expression of the form `new ValType()`, where `ValType` is a value type, such as an `enum` or a `struct`;
- an expression of the form `default(ValType)`, where `ValType` is a value type.

Optional parameters are defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters. Comma-separated gaps in the argument list aren't supported. For example, in

the following code, instance method `ExampleMethod` is defined with one required and two optional parameters.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

The following call to `ExampleMethod` causes a compiler error, because an argument is provided for the third parameter but not for the second.

```
//anExample.ExampleMethod(3, ,4);
```

However, if you know the name of the third parameter, you can use a named argument to accomplish the task.

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense uses brackets to indicate optional parameters, as shown in the following illustration:

```
anExample.ExampleMethod(
    void ExampleClass.ExampleMethod(int required,
        [string optionalstr = "default string"],
        [int optionalint = 10])
```

NOTE

You can also declare optional parameters by using the .NET [OptionalAttribute](#) class. `OptionalAttribute` parameters do not require a default value.

Example

In the following example, the constructor for `ExampleClass` has one parameter, which is optional. Instance method `ExampleMethod` has one required parameter, `required`, and two optional parameters, `optionalstr` and `optionalint`. The code in `Main` shows the different ways in which the constructor and method can be invoked.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();
```

```

// You cannot leave a gap in the provided arguments.
//anExample.ExampleMethod(3, ,4);
//anExample.ExampleMethod(3, 4);

// You can use a named parameter to make the previous
// statement work.
anExample.ExampleMethod(3, optionalint: 4);
}

}

class ExampleClass
{
    private string _name;

    // Because the parameter for the constructor, name, has a default
    // value assigned to it, it is optional.
    public ExampleClass(string name = "Default name")
    {
        _name = name;
    }

    // The first parameter, required, has no default value assigned
    // to it. Therefore, it is not optional. Both optionalstr and
    // optionalint have default values assigned to them. They are optional.
    public void ExampleMethod(int required, string optionalstr = "default string",
        int optionalint = 10)
    {
        Console.WriteLine(
            $"({_name}): {required}, {optionalstr}, and {optionalint}.");
    }
}

// The output from this example is the following:
// Default name: 1, One, and 1.
// Default name: 2, Two, and 10.
// Default name: 3, default string, and 10.
// Provided name: 1, One, and 1.
// Provided name: 2, Two, and 10.
// Provided name: 3, default string, and 10.
// Default name: 3, default string, and 4.
}

```

The preceding code shows a number of examples where optional parameters aren't applied correctly. The first illustrates that an argument must be supplied for the first parameter, which is required.

COM Interfaces

Named and optional arguments, along with support for dynamic objects, greatly improve interoperability with COM APIs, such as Office Automation APIs.

For example, the [AutoFormat](#) method in the Microsoft Office Excel [Range](#) interface has seven parameters, all of which are optional. These parameters are shown in the following illustration:

```

excelApp.get_Range("A1", "B4").AutoFormat(
    dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
    [object Number = Type.Missing], [object Font = Type.Missing],
    [object Alignment = Type.Missing], [object Border = Type.Missing],
    [object Pattern = Type.Missing], [object Width = Type.Missing])
}

```

In C# 3.0 and earlier versions, an argument is required for each parameter, as shown in the following example.

```

// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);

```

However, you can greatly simplify the call to `AutoFormat` by using named and optional arguments, introduced in C# 4.0. Named and optional arguments enable you to omit the argument for an optional parameter if you don't want to change the parameter's default value. In the following call, a value is specified for only one of the seven parameters.

```

// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );

```

For more information and examples, see [How to use named and optional arguments in Office programming](#) and [How to access Office interop objects by using C# features](#).

Overload Resolution

Use of named and optional arguments affects overload resolution in the following ways:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that doesn't have optional parameters for which arguments were omitted in the call. Overload resolution generally prefers candidates that have fewer parameters.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

How to use named and optional arguments in Office programming (C# Programming Guide)

3/6/2021 • 5 minutes to read • [Edit Online](#)

Named arguments and optional arguments, introduced in C# 4, enhance convenience, flexibility, and readability in C# programming. In addition, these features greatly facilitate access to COM interfaces such as the Microsoft Office automation APIs.

In the following example, method `ConvertToTable` has sixteen parameters that represent characteristics of a table, such as number of columns and rows, formatting, borders, fonts, and colors. All sixteen parameters are optional, because most of the time you do not want to specify particular values for all of them. However, without named and optional arguments, a value or a placeholder value has to be provided for each parameter. With named and optional arguments, you specify values only for the parameters that are required for your project.

You must have Microsoft Office Word installed on your computer to complete these procedures.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Templates Categories** pane, expand **Visual C#**, and then click **Windows**.
4. Look in the top of the **Templates** pane to make sure that **.NET Framework 4** appears in the **Target Framework** box.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

The new project appears in **Solution Explorer**.

To add a reference

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **.NET** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list.
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the *Program.cs* file and then click **View Code**.

2. Add the following `using` directives to the top of the code file:

```
using Word = Microsoft.Office.Interop.Word;
```

To display text in a Word document

1. In the `Program` class in `Program.cs`, add the following method to create a Word application and a Word document. The `Add` method has four optional parameters. This example uses their default values. Therefore, no arguments are necessary in the calling statement.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Add the following code at the end of the method to define where to display text in the document, and what text to display:

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

To run the application

1. Add the following statement to Main:

```
DisplayInWord();
```

2. Press CTRL+F5 to run the project. A Word document appears that contains the specified text.

To change the text to a table

1. Use the `ConvertToTable` method to enclose the text in a table. The method has sixteen optional parameters. IntelliSense encloses optional parameters in brackets, as shown in the following illustration.

```
range.ConvertToTable()

Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Named and optional arguments enable you to specify values for only the parameters that you want to change. Add the following code to the end of method `DisplayInWord` to create a simple table. The argument specifies that the commas in the text string in `range` separate the cells of the table.

```
// Convert to a simple table. The table will have a single row with  
// three columns.  
range.ConvertToTable(Separator: ",");
```

In earlier versions of C#, the call to `Convert.ToTable` requires a reference argument for each parameter, as shown in the following code:

```
// Call to Convert.ToTable in Visual C# 2008 or earlier. This code  
// is not part of the solution.  
var missing = Type.Missing;  
object separator = ",";  
range.Convert.ToTable(ref separator, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing);
```

2. Press CTRL+F5 to run the project.

To experiment with other parameters

1. To change the table so that it has one column and three rows, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5.

```
range.Convert.ToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. To specify a predefined format for the table, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5. The format can be any of the [WdTableFormat](#) constants.

```
range.Convert.ToTable(Separator: ",", AutoFit: true, NumColumns: 1,  
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Example

The following code includes the full example:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}

```

See also

- [Named and Optional Arguments](#)

Constructors (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Whenever a [class](#) or [struct](#) is created, its constructor is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read. For more information and examples, see [Using Constructors](#) and [Instance Constructors](#).

Parameterless constructors

If you don't provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values as listed in the [Default values of C# types](#) article. If you don't provide a constructor for your struct, C# relies on an *implicit parameterless constructor* to automatically initialize each field to its default value. For more information and examples, see [Instance constructors](#).

Constructor syntax

A constructor is a method whose name is the same as the name of its type. Its method signature includes only the method name and its parameter list; it does not include a return type. The following example shows the constructor for a class named `Person`.

```
public class Person
{
    private string last;
    private string first;

    public Person(string lastName, string firstName)
    {
        last = lastName;
        first = firstName;
    }

    // Remaining implementation of Person class.
}
```

If a constructor can be implemented as a single statement, you can use an [expression body definition](#). The following example defines a `Location` class whose constructor has a single string parameter named `name`. The expression body definition assigns the argument to the `locationName` field.

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Static constructors

The previous examples have all shown instance constructors, which create a new object. A class or struct can also have a static constructor, which initializes static members of the type. Static constructors are parameterless. If you don't provide a static constructor to initialize static fields, the C# compiler initializes static fields to their default value as listed in the [Default values of C# types](#) article.

The following example uses a static constructor to initialize a static field.

```
public class Adult : Person
{
    private static int minimumAge;

    public Adult(string lastName, string firstName) : base(lastName, firstName)
    { }

    static Adult()
    {
        minimumAge = 18;
    }

    // Remaining implementation of Adult class.
}
```

You can also define a static constructor with an expression body definition, as the following example shows.

```
public class Child : Person
{
    private static int maximumAge;

    public Child(string lastName, string firstName) : base(lastName, firstName)
    { }

    static Child() => maximumAge = 18;

    // Remaining implementation of Child class.
}
```

For more information and examples, see [Static Constructors](#).

In This Section

[Using Constructors](#)

[Instance Constructors](#)

[Private Constructors](#)

[Static Constructors](#)

[How to write a copy constructor](#)

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Finalizers](#)
- [static](#)
- [Why Do Initializers Run In The Opposite Order As Constructors? Part One](#)

Using Constructors (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

When a [class](#) or [struct](#) is created, its constructor is called. Constructors have the same name as the class or [struct](#), and they usually initialize the data members of the new object.

In the following example, a class named `Taxi` is defined by using a simple constructor. This class is then instantiated with the `new` operator. The `Taxi` constructor is invoked by the `new` operator immediately after memory is allocated for the new object.

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

A constructor that takes no parameters is called a *parameterless constructor*. Parameterless constructors are invoked whenever an object is instantiated by using the `new` operator and no arguments are provided to `new`. For more information, see [Instance Constructors](#).

Unless the class is [static](#), classes without constructors are given a public parameterless constructor by the C# compiler in order to enable class instantiation. For more information, see [Static Classes and Static Class Members](#).

You can prevent a class from being instantiated by making the constructor private, as follows:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

For more information, see [Private Constructors](#).

Constructors for [struct](#) types resemble class constructors, but [structs](#) cannot contain an explicit parameterless constructor because one is provided automatically by the compiler. This constructor initializes each field in the [struct](#) to the [default value](#). However, this parameterless constructor is only invoked if the [struct](#) is instantiated with `new`. For example, this code uses the parameterless constructor for [Int32](#), so that you are assured that the integer is initialized:

```
int i = new int();
Console.WriteLine(i);
```

The following code, however, causes a compiler error because it does not use `new`, and because it tries to use an object that has not been initialized:

```
int i;
Console.WriteLine(i);
```

Alternatively, objects based on `structs` (including all built-in numeric types) can be initialized or assigned and then used as in the following example:

```
int a = 44; // Initialize the value type...
int b;
b = 33; // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

So calling the parameterless constructor for a value type is not required.

Both classes and `structs` can define constructors that take parameters. Constructors that take parameters must be called through a `new` statement or a `base` statement. Classes and `structs` can also define multiple constructors, and neither is required to define a parameterless constructor. For example:

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

This class can be created by using either of the following statements:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

A constructor can use the `base` keyword to call the constructor of a base class. For example:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

In this example, the constructor for the base class is called before the block for the constructor is executed. The `base` keyword can be used with or without parameters. Any parameters to the constructor can be used as parameters to `base`, or as part of an expression. For more information, see [base](#).

In a derived class, if a base-class constructor is not called explicitly by using the `base` keyword, the parameterless constructor, if there is one, is called implicitly. This means that the following constructor declarations are effectively the same:

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
: base()
{
    //Add further instructions here.
}
```

If a base class does not offer a parameterless constructor, the derived class must make an explicit call to a base constructor by using `base`.

A constructor can invoke another constructor in the same object by using the `this` keyword. Like `base`, `this` can be used with or without parameters, and any parameters in the constructor are available as parameters to `this`, or as part of an expression. For example, the second constructor in the previous example can be rewritten using `this`:

```
public Employee(int weeklySalary, int numberOfWeeks)
: this(weeklySalary * numberOfWeeks)
{
}
```

The use of the `this` keyword in the previous example causes this constructor to be called:

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Constructors can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) or [private protected](#). These access modifiers define how users of the class can construct the class. For more information, see [Access Modifiers](#).

A constructor can be declared static by using the `static` keyword. Static constructors are called automatically, immediately before any static fields are accessed, and are generally used to initialize static class members. For more information, see [Static Constructors](#).

C# Language Specification

For more information, see [Instance constructors](#) and [Static constructors](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Constructors](#)
- [Finalizers](#)

Instance Constructors (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

Instance constructors are used to create and initialize any instance member variables when you use the `new` expression to create an object of a [class](#). To initialize a [static](#) class, or static variables in a non-static class, you define a static constructor. For more information, see [Static Constructors](#).

The following example shows an instance constructor:

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

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

NOTE

For clarity, this class contains public fields. The use of public fields is not a recommended programming practice because it allows any method anywhere in a program unrestricted and unverified access to an object's inner workings. Data members should generally be private, and should be accessed only through class methods and properties.

This instance constructor is called whenever an object based on the `Coords` class is created. A constructor like this one, which takes no arguments, is called a *parameterless constructor*. However, it is often useful to provide additional constructors. For example, we can add a constructor to the `Coords` class that allows us to specify the initial values for the data members:

```
// A constructor with two arguments.
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

This allows `Coords` objects to be created with default or specific initial values, like this:

```
var p1 = new Coords();
var p2 = new Coords(5, 3);
```

If a class does not have a constructor, a parameterless constructor is automatically generated and default values are used to initialize the object fields. For example, an `int` is initialized to 0. For information about the type default values, see [Default values of C# types](#). Therefore, because the `Coords` class parameterless constructor initializes all data members to zero, it can be removed altogether without changing how the class works. A complete example using multiple constructors is provided in Example 1 later in this topic, and an example of an automatically generated constructor is provided in Example 2.

Instance constructors can also be used to call the instance constructors of base classes. The class constructor can invoke the constructor of the base class through the initializer, as follows:

```
class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}
```

In this example, the `Circle` class passes values representing radius and height to the constructor provided by `Shape` from which `Circle` is derived. A complete example using `Shape` and `Circle` appears in this topic as Example 3.

Example 1

The following example demonstrates a class with two class constructors, one without arguments and one with two arguments.

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

    // Default constructor.
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments.
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method.
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        var p1 = new Coords();
        var p2 = new Coords(5, 3);

        // Display the results using the overridden ToString method.
        Console.WriteLine($"Coords #1 at {p1}");
        Console.WriteLine($"Coords #2 at {p2}");
        Console.ReadKey();
    }
}
/* Output:
Coords #1 at (0,0)
Coords #2 at (5,3)
*/
```

Example 2

In this example, the class `Person` does not have any constructors, in which case, a parameterless constructor is automatically provided and the fields are initialized to their default values.

```
using System;

public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        var person = new Person();

        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Name: , Age: 0
```

Notice that the default value of `age` is `0` and the default value of `name` is `null`.

Example 3

The following example demonstrates using the base class initializer. The `Circle` class is derived from the general class `Shape`, and the `Cylinder` class is derived from the `Circle` class. The constructor on each derived class is using its base class initializer.

```

using System;

abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

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

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }

    public override double Area()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Area of the circle = 19.63
   Area of the cylinder = 86.39
*/

```

For more examples on invoking the base class constructors, see [virtual](#), [override](#), and [base](#).

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Constructors](#)
- [Finalizers](#)
- [static](#)

Private Constructors (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class. For example:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

The declaration of the empty constructor prevents the automatic generation of a parameterless constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the [private](#) modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the [Math](#) class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static. For more information see [Static Classes and Static Class Members](#).

Example

The following is an example of a class using a private constructor.

```
public class Counter
{
    private Counter() { }

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter(); // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Notice that if you uncomment the following statement from the example, it will generate an error because the constructor is inaccessible because of its protection level:

C# Language Specification

For more information, see [Private constructors](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Constructors](#)
- [Finalizers](#)
- [private](#)
- [public](#)

Static Constructors (C# Programming Guide)

4/1/2021 • 4 minutes to read • [Edit Online](#)

A static constructor is used to initialize any [static](#) data, or to perform a particular action that needs to be performed only once. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Remarks

Static constructors have the following properties:

- A static constructor doesn't take access modifiers or have parameters.
- A class or struct can only have one static constructor.
- Static constructors cannot be inherited or overloaded.
- A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically.
- The user has no control on when the static constructor is executed in the program.
- A static constructor is called automatically. It initializes the [class](#) before the first instance is created or any static members are referenced. A static constructor runs before an instance constructor. A type's static constructor is called when a static method assigned to an event or a delegate is invoked and not when it is assigned. If static field variable initializers are present in the class of the static constructor, they're executed in the textual order in which they appear in the class declaration. The initializers run immediately prior to the execution of the static constructor.
- If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in [Default values of C# types](#).
- If a static constructor throws an exception, the runtime doesn't invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain. Most commonly, a [TypeInitializationException](#) exception is thrown when a static constructor is unable to instantiate a type or for an unhandled exception occurring within a static constructor. For static constructors that aren't explicitly defined in source code, troubleshooting may require inspection of the intermediate language (IL) code.
- The presence of a static constructor prevents the addition of the [BeforeFieldInit](#) type attribute. This limits runtime optimization.

- A field declared as `static readonly` may only be assigned as part of its declaration or in a static constructor. When an explicit static constructor isn't required, initialize static fields at declaration rather than through a static constructor for better runtime optimization.
- The runtime calls a static constructor no more than once in a single application domain. That call is made in a locked region based on the specific type of the class. No additional locking mechanisms are needed in the body of a static constructor. To avoid the risk of deadlocks, don't block the current thread in static constructors and initializers. For example, don't wait on tasks, threads, wait handles or events, don't acquire locks, and don't execute blocking parallel operations such as parallel loops, `Parallel.Invoke` and Parallel LINQ queries.

NOTE

Though not directly accessible, the presence of an explicit static constructor should be documented to assist with troubleshooting initialization exceptions.

Usage

- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the `LoadLibrary` method.
- Static constructors are also a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile time via type-parameter constraints.

Example

In this example, class `Bus` has a static constructor. When the first instance of `Bus` is created (`bus1`), the static constructor is invoked to initialize the class. The sample output verifies that the static constructor runs only one time, even though two instances of `Bus` are created, and that it runs before the instance constructor runs.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }
}
```

```

// Instance method.
public void Drive()
{
    TimeSpan elapsedTime = DateTime.Now - globalStartTime;

    // For demonstration purposes we treat milliseconds as minutes to simulate
    // actual bus times. Do not do this in your actual bus schedule program!
    Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
                      this.RouteNumber,
                      elapsedTime.Milliseconds,
                      globalStartTime.ToShortTimeString());
}

}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Sample output:
   Static constructor sets global start time to 3:57:08 PM.
   Bus #71 is created.
   Bus #72 is created.
   71 is starting its route 6.00 minutes after global start time 3:57 PM.
   72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/

```

C# language specification

For more information, see the [Static constructors](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Constructors](#)
- [Static Classes and Static Class Members](#)
- [Finalizers](#)
- [Constructor Design Guidelines](#)
- [Security Warning - CA2121: Static constructors should be private](#)

How to write a copy constructor (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

C# [records](#) provide a copy constructor for objects, but for classes you have to write one yourself.

Example

In the following example, the `Person class` defines a copy constructor that takes, as its argument, an instance of `Person`. The values of the properties of the argument are assigned to the properties of the new instance of `Person`. The code contains an alternative copy constructor that sends the `Name` and `Age` properties of the instance that you want to copy to the instance constructor of the class.

```

using System;

class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41

```

See also

- [ICloneable](#)
- [Records](#)
- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Constructors](#)
- [Finalizers](#)

Finalizers (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

Finalizers (which are also called **destructors**) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the `Car` class.

```
class Car
{
    ~Car() // finalizer
    {
        // cleanup statements...
    }
}
```

A finalizer can also be implemented as an expression body definition, as the following example shows.

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

The finalizer implicitly calls [Finalize](#) on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

This design means that the `Finalize` method is called recursively for all instances in the inheritance chain, from

the most-derived to the least-derived.

NOTE

Empty finalizers should not be used. When a class contains a finalizer, an entry is created in the `Finalize` queue. When the finalizer is called, the garbage collector is invoked to process the queue. An empty finalizer just causes a needless loss of performance.

The programmer has no control over when the finalizer is called; the garbage collector decides when to call it. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object.

In .NET Framework applications (but not in .NET Core applications), finalizers are also called when the program exits.

It's possible to force garbage collection by calling `Collect`, but most of the time, this call should be avoided because it may create performance issues.

Using finalizers to release resources

In general, C# does not require as much memory management on the part of the developer as languages that don't target a runtime with garbage collection. This is because the .NET garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources, such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the `Finalize` method of the object.

Explicit release of resources

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. To release the resource, implement a `Dispose` method from the `IDisposable` interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the `Dispose` method fails.

For more information about cleaning up resources, see the following articles:

- [Cleaning Up Unmanaged Resources](#)
- [Implementing a Dispose Method](#)
- [using Statement](#)

Example

The following example creates three classes that make a chain of inheritance. The class `First` is the base class, `Second` is derived from `First`, and `Third` is derived from `Second`. All three have finalizers. In `Main`, an instance of the most-derived class is created. When the program runs, notice that the finalizers for the three classes are called automatically, and in order, from the most-derived to the least-derived.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }
}
/* Output (to VS Output Window):
   Third's finalizer is called.
   Second's finalizer is called.
   First's finalizer is called.
*/
```

C# language specification

For more information, see the [Destructors](#) section of the C# language specification.

See also

- [IDisposable](#)
- [C# Programming Guide](#)
- [Constructors](#)
- [Garbage Collection](#)

Object and Collection Initializers (C# Programming Guide)

11/2/2020 • 8 minutes to read • [Edit Online](#)

C# lets you instantiate an object or collection and perform member assignments in a single statement.

Object initializers

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments (and parentheses syntax). The following example shows how to use an object initializer with a named type, `Cat` and how to invoke the parameterless constructor. Note the use of auto-implemented properties in the `Cat` class. For more information, see [Auto-Implemented Properties](#).

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

The object initializers syntax allows you to create an instance, and after that it assigns the newly created object, with its assigned properties, to the variable in the assignment.

Starting with C# 6, object initializers can set indexers, in addition to assigning fields and properties. Consider this basic `Matrix` class:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

You could initialize the identity matrix with the following code:

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Any accessible indexer that contains an accessible setter can be used as one of the expressions in an object initializer, regardless of the number or types of arguments. The index arguments form the left side of the assignment, and the value is the right side of the expression. For example, these are all valid if `IndexersExample` has the appropriate indexers:

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

For the preceding code to compile, the `IndexersExample` type must have the following members:

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

Object Initializers with anonymous types

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. Query expressions make frequent use of [anonymous types](#), which can only be initialized by using an object initializer, as shown in the following declaration.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Anonymous types enable the `select` clause in a LINQ query expression to transform objects of the original sequence into objects whose value and shape may differ from the original. This is useful if you want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (`p`) contains many fields and methods, and that you are only interested in creating a sequence of objects that contain the product name and the unit price.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

When this query is executed, the `productInfos` variable will contain a sequence of objects that can be accessed in a `foreach` statement as shown in this example:

```
foreach(var p in productInfos){...}
```

Each object in the new anonymous type has two public properties that receive the same names as the properties or fields in the original object. You can also rename a field when you are creating an anonymous type; the following example renames the `UnitPrice` field to `Price`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

Collection initializers

Collection initializers let you specify one or more element initializers when you initialize a collection type that implements `IEnumerable` and has `Add` with the appropriate signature as an instance method or an extension method. The element initializers can be a simple value, an expression, or an object initializer. By using a collection initializer, you do not have to specify multiple calls; the compiler adds the calls automatically.

The following example shows two simple collection initializers:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

The following collection initializer uses object initializers to initialize objects of the `Cat` class defined in a previous example. Note that the individual object initializers are enclosed in braces and separated by commas.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

You can specify `null` as an element in a collection initializer if the collection's `Add` method allows it.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

You can specify indexed elements if the collection supports read / write indexing.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

The preceding sample generates code that calls the `Item[TKey]` to set the values. Before C# 6, you could initialize dictionaries and other associative containers using the following syntax. Notice that instead of indexer syntax,

with parentheses and an assignment, it uses an object with multiple values:

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

This initializer example calls [Add\(TKey, TValue\)](#) to add the three items into the dictionary. These two different ways to initialize associative collections have slightly different behavior because of the method calls the compiler generates. Both variants work with the `Dictionary` class. Other types may only support one or the other based on their public API.

Examples

The following example combines the concepts of object and collection initializers.

```

public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
}

```

The following example shows an object that implements [IEnumerable](#) and contains an [Add](#) method with multiple parameters. It uses a collection initializer with multiple elements per item in the list that correspond to the signature of the [Add](#) method.

```

public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new List<string>();
        public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
                        string street, string city,
                        string state, string zipcode) => internalList.Add(
                        $"'{firstname} {lastname}'
{street}
{city}, {state} {zipcode}"
                    );
    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)
        {
            Console.WriteLine("\r\n" + addressEntry);
        }
    }

/*
 * Prints:

    Address Entries:

    John Doe
    123 Street
    Topeka, KS 00000

    Jane Smith
    456 Street
    Topeka, KS 00000
*/
}

```

Add methods can use the `params` keyword to take a variable number of arguments, as shown in the following example. This example also demonstrates the custom implementation of an indexer to initialize a collection using indexes.

```

public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();
    }
}

```

```

public List< TValue> this[TKey key]
{
    get => internalDictionary[key];
    set => Add(key, value);
}

public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable< TValue>)values);

public void Add(TKey key, IEnumerable< TValue> values)
{
    if (!internalDictionary.TryGetValue(key, out List< TValue> storedValues))
        internalDictionary.Add(key, storedValues = new List< TValue>());

    storedValues.AddRange(values);
}
}

public static void Main()
{
    RudimentaryMultiValuedDictionary< string, string > rudimentaryMultiValuedDictionary1
        = new RudimentaryMultiValuedDictionary< string, string >()
    {
        {"Group1", "Bob", "John", "Mary" },
        {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
    };
    RudimentaryMultiValuedDictionary< string, string > rudimentaryMultiValuedDictionary2
        = new RudimentaryMultiValuedDictionary< string, string >()
    {
        ["Group1"] = new List< string >() { "Bob", "John", "Mary" },
        ["Group2"] = new List< string >() { "Eric", "Emily", "Debbie", "Jesse" }
    };
    RudimentaryMultiValuedDictionary< string, string > rudimentaryMultiValuedDictionary3
        = new RudimentaryMultiValuedDictionary< string, string >()
    {
        {"Group1", new string []{ "Bob", "John", "Mary" } },
        { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
    };

    Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

    foreach (KeyValuePair< string, List< string >> group in rudimentaryMultiValuedDictionary1)
    {
        Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }

    Console.WriteLine("\\r\\nUsing second multi-valued dictionary created with a collection initializer
using indexing:");

    foreach (KeyValuePair< string, List< string >> group in rudimentaryMultiValuedDictionary2)
    {
        Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }

    Console.WriteLine("\\r\\nUsing third multi-valued dictionary created with a collection initializer
using indexing:");

    foreach (KeyValuePair< string, List< string >> group in rudimentaryMultiValuedDictionary3)
    {
        Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");
    }
}

```

```

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }

/*
 * Prints:

Using first multi-valued dictionary created with a collection initializer:

Members of group Group1:
Bob
John
Mary

Members of group Group2:
Eric
Emily
Debbie
Jesse

Using second multi-valued dictionary created with a collection initializer using indexing:

Members of group Group1:
Bob
John
Mary

Members of group Group2:
Eric
Emily
Debbie
Jesse

Using third multi-valued dictionary created with a collection initializer using indexing:

Members of group Group1:
Bob
John
Mary

Members of group Group2:
Eric
Emily
Debbie
Jesse
*/
}

```

See also

- [C# Programming Guide](#)
- [LINQ in C#](#)
- [Anonymous Types](#)

How to initialize objects by using an object initializer (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type.

The following examples show how to use object initializers with named objects. The compiler processes object initializers by first accessing the parameterless instance constructor and then processing the member initializations. Therefore, if the parameterless constructor is declared as `private` in the class, object initializers that require public access will fail.

You must use an object initializer if you're defining an anonymous type. For more information, see [How to return subsets of element properties in a query](#).

Example

The following example shows how to initialize a new `StudentName` type by using object initializers. This example sets properties in the `StudentName` type:

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The parameterless constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the parameterless constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
    }
}
```

```

        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig 0
    // Craig 0
    // 183
    // Craig 116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + " " + ID;
    }
}

```

Object initializers can be used to set indexers in an object. The following example defines a `BaseballTeam` class that uses an indexer to get and set players at different positions. The initializer can assign players, based on the abbreviation for the position, or the number used for each position baseball scorecards:

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }

        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

See also

- [C# Programming Guide](#)
- [Object and Collection Initializers](#)

How to initialize a dictionary with a collection initializer (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

A `Dictionary<TKey,TValue>` contains a collection of key/value pairs. Its `Add` method takes two parameters, one for the key and one for the value. One way to initialize a `Dictionary<TKey,TValue>`, or any collection whose `Add` method takes multiple parameters, is to enclose each set of parameters in braces as shown in the following example. Another option is to use an index initializer, also shown in the following example.

Example

In the following code example, a `Dictionary<TKey,TValue>` is initialized with instances of type `StudentName`. The first initialization uses the `Add` method with two arguments. The compiler generates a call to `Add` for each of the pairs of `int` keys and `StudentName` values. The second uses a public read / write indexer method of the `Dictionary` class:

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 },
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName}
{students2[index].LastName}");
        }
    }
}
```

Note the two pairs of braces in each element of the collection in the first declaration. The innermost braces

enclose the object initializer for the `StudentName`, and the outermost braces enclose the initializer for the key/value pair that will be added to the `students Dictionary< TKey, TValue >`. Finally, the whole collection initializer for the dictionary is enclosed in braces. In the second initialization, the left side of the assignment is the key and the right side is the value, using an object initializer for `StudentName`.

See also

- [C# Programming Guide](#)
- [Object and Collection Initializers](#)

Nested Types (C# Programming Guide)

3/9/2021 • 2 minutes to read • [Edit Online](#)

A type defined within a [class](#), [struct](#), or [interface](#) is called a nested type. For example

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Regardless of whether the outer type is a class, interface, or struct, nested types default to [private](#); they are accessible only from their containing type. In the previous example, the `Nested` class is inaccessible to external types.

You can also specify an [access modifier](#) to define the accessibility of a nested type, as follows:

- Nested types of a [class](#) can be [public](#), [protected](#), [internal](#), [protected internal](#), [private](#) or [private protected](#).

However, defining a `protected`, `protected internal` or `private protected` nested class inside a [sealed class](#) generates compiler warning [CS0628](#), "new protected member declared in sealed class."

Also be aware that making a nested type externally visible violates the code quality rule [CA1034](#) "Nested types should not be visible".

- Nested types of a [struct](#) can be [public](#), [internal](#), or [private](#).

The following example makes the `Nested` class public:

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

The nested, or inner, type can access the containing, or outer, type. To access the containing type, pass it as an argument to the constructor of the nested type. For example:

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }

        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

A nested type has access to all of the members that are accessible to its containing type. It can access private and protected members of the containing type, including any inherited protected members.

In the previous declaration, the full name of class `Nested` is `Container.Nested`. This is the name used to create a new instance of the nested class, as follows:

```
Container.Nested nest = new Container.Nested();
```

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Access Modifiers](#)
- [Constructors](#)
- [CA1034 rule](#)

Partial Classes and Methods (C# Programming Guide)

3/25/2021 • 6 minutes to read • [Edit Online](#)

It is possible to split the definition of a [class](#), a [struct](#), an [interface](#) or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- To split a class definition, use the [partial](#) keyword modifier, as shown here:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

The [partial](#) keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the [partial](#) keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as [public](#), [private](#), and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

NOTE

The [partial](#) modifier is not available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

```
class Container
{
    partial class Nested
    {
        void Test() { }

        partial class Nested
        {
            void Test2() { }
        }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They are equivalent to the following declarations:

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They are equivalent to the following declarations:

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

```
public partial class A { }
//public class A { } // Error, must also be marked partial
```

- The `partial` modifier can only appear immediately before the keywords `class`, `struct`, or `interface`.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
 - `public`
 - `private`
 - `protected`
 - `internal`
 - `abstract`
 - `sealed`
 - base class
 - `new` modifier (nested parts)
 - generic constraints

For more information, see [Constraints on Type Parameters](#).

Example 1

Description

In the following example, the fields and the constructor of the class, `Coords`, are declared in one partial class definition, and the member, `PrintCoords`, is declared in another partial class definition.

Code

```

public partial class Coords
{
    private int x;
    private int y;

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

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15

```

Example 2

Description

The following example shows that you can also develop partial structs and interfaces.

Code

```

partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}

```

Partial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An implementation can be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time. Implementation may be required depending on method signature.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
    // method body
}
```

- Partial method declarations must begin with the contextual keyword [partial](#).
- Partial method signatures in both parts of the partial type must match.
- Partial methods can have [static](#) and [unsafe](#) modifiers.
- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
- You can make a [delegate](#) to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

A partial method isn't required to have an implementation in the following cases:

- It doesn't have any accessibility modifiers (including the default [private](#)).
- It returns [void](#).
- It doesn't have any [out](#) parameters.
- It doesn't have any of the following modifiers [virtual](#), [override](#), [sealed](#), [new](#), or [extern](#).

Any method that doesn't conform to all those restrictions (for example, `public virtual partial void` method), must provide an implementation.

C# Language Specification

For more information, see [Partial types](#) in the [C# Language Specification](#). The language specification is the

definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes](#)
- [Structure types](#)
- [Interfaces](#)
- [partial \(Type\)](#)

Anonymous Types (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the `new` operator together with an object initializer. For more information about object initializers, see [Object and Collection Initializers](#).

The following example shows an anonymous type that is initialized with two properties named `Amount` and `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Anonymous types typically are used in the `select` clause of a query expression to return a subset of the properties from each object in the source sequence. For more information about queries, see [LINQ in C#](#).

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be `null`, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named `Product`. Class `Product` includes `Color` and `Price` properties, together with other properties that you are not interested in. Variable `products` is a collection of `Product` objects. The anonymous type declaration starts with the `new` keyword. The declaration initializes a new type that uses only two properties from `Product`. This causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You must provide a name for a property that is being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are `Color` and `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using `var`. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type. For more information about `var`, see [Implicitly Typed Local Variables](#).

You can create an array of anonymously typed elements by combining an implicitly typed local variable and an

implicitly typed array, as shown in the following example.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Remarks

Anonymous types are [class](#) types that derive directly from [object](#), and that cannot be cast to any type except [object](#). The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

You cannot declare a field, a property, an event, or the return type of a method as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type [object](#). However, doing this defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Because the [Equals](#) and [GetHashCode](#) methods on anonymous types are defined in terms of the [Equals](#) and [GetHashCode](#) methods of the properties, two instances of the same anonymous type are equal only if all their properties are equal.

See also

- [C# Programming Guide](#)
- [Object and Collection Initializers](#)
- [Getting Started with LINQ in C#](#)
- [LINQ in C#](#)

How to return subsets of element properties in a query (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Use an anonymous type in a query expression when both of these conditions apply:

- You want to return only some of the properties of each source element.
- You do not have to store the query results outside the scope of the method in which the query is executed.

If you only want to return one property or field from each source element, then you can just use the dot operator in the `select` clause. For example, to return only the `ID` of each `student`, write the `select` clause as follows:

```
select student.ID;
```

Example

The following example shows how to use an anonymous type to return only a subset of the properties of each source element that matches the specified condition.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Note that the anonymous type uses the source element's names for its properties if no names are specified. To give new names to the properties in the anonymous type, write the `select` statement as follows:

```
select new { First = student.FirstName, Last = student.LastName };
```

If you try this in the previous example, then the `Console.WriteLine` statement must also change:

```
Console.WriteLine(student.First + " " + student.Last);
```

Compiling the Code

To run this code, copy and paste the class into a C# console application with a `using` directive for `System.Linq`.

See also

- [C# Programming Guide](#)
- [Anonymous Types](#)
- [LINQ in C#](#)

Interfaces (C# Programming Guide)

3/23/2021 • 4 minutes to read • [Edit Online](#)

An interface contains definitions for a group of related functionalities that a non-abstract [class](#) or a [struct](#) must implement. An interface may define [static](#) methods, which must have an implementation. Beginning with C# 8.0, an interface may define a default implementation for members. An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the [interface](#) keyword as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of an interface must be a valid C# [identifier name](#). By convention, interface names begin with a capital [I](#).

Any class or struct that implements the [IEquatable<T>](#) interface must contain a definition for an [Equals](#) method that matches the signature that the interface specifies. As a result, you can count on a class that implements [IEquatable<T>](#) to contain an [Equals](#) method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of [IEquatable<T>](#) doesn't provide an implementation for [Equals](#). A class or struct can implement multiple interfaces, but a class can only inherit from a single class.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members](#).

Interfaces can contain instance methods, properties, events, indexers, or any combination of those four member types. Interfaces may contain static constructors, fields, constants, or operators. For links to examples, see [Related Sections](#).

An interface can't contain instance fields, instance constructors, or finalizers. Interface members are public by default, and you can explicitly specify accessibility modifiers, such as [public](#), [protected](#), [internal](#), [private](#), [protected internal](#), or [private protected](#). A [private](#) member must have a default implementation.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface declares but doesn't provide a default implementation for. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the [IEquatable<T>](#) interface. The implementing class, [Car](#), must provide an implementation of the [Equals](#) method.

```

public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}

```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a `get` accessor. The class that implements the interface can declare the same property with both a `get` and `set` accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see [Explicit Interface Implementation](#) and [Interface Properties](#).

Interfaces can inherit from one or more interfaces. The derived interface inherits the members from its base interfaces. A class that implements a derived interface must implement all members in the derived interface, including all members of the derived interface's base interfaces. That class may be implicitly converted to the derived interface or any of its base interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation. When interfaces declare a default implementation of a method, any class implementing that interface inherits that implementation. Implementations defined in interfaces are virtual and the implementing class may override that implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see [Polymorphism](#).

Interfaces summary

An interface has the following properties:

- An interface is typically like an abstract base class with only abstract members. Any class or struct that implements the interface must implement all its members. Optionally, an interface may define default implementations for some or all of its members. For more information, see [default interface methods](#).
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

Related Sections

- [Interface Properties](#)
- [Indexers in Interfaces](#)
- [How to implement interface events](#)
- [Classes and Structs](#)

- [Inheritance](#)
- [Interfaces](#)
- [Methods](#)
- [Polymorphism](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Properties](#)
- [Events](#)
- [Indexers](#)

See also

- [C# Programming Guide](#)
- [Inheritance](#)
- [Identifier names](#)

Explicit Interface Implementation (C# Programming Guide)

3/26/2021 • 2 minutes to read • [Edit Online](#)

If a [class](#) implements two interfaces that contain a member with the same signature, then implementing that member on the class will cause both interfaces to use that member as their implementation. In the following example, all the calls to `Paint` invoke the same method. This first sample defines the types:

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

The following sample calls the methods:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();
// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

But you might not want the same implementation to be called for both interfaces. To call a different implementation depending on which interface is in use, you can implement an interface member explicitly. An explicit interface implementation is a class member that is only called through the specified interface. Name the class member by prefixing it with the name of the interface and a period. For example:

```

public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}

```

The class member `IControl.Paint` is only available through the `IControl` interface, and `ISurface.Paint` is only available through `ISurface`. Both method implementations are separate, and neither are available directly on the class. For example:

```

// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint(); // Compiler error.

IControl c = obj;
c.Paint(); // Calls IControl.Paint on SampleClass.

ISurface s = obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint

```

Explicit implementation is also used to resolve cases where two interfaces each declare different members of the same name such as a property and a method. To implement both interfaces, a class has to use explicit implementation either for the property `P`, or the method `P`, or both, to avoid a compiler error. For example:

```

interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}

```

An explicit interface implementation doesn't have an access modifier since it isn't accessible as a member of the type it's defined in. Instead, it's only accessible when called through an instance of the interface. If you specify an access modifier for an explicit interface implementation, you get compiler error [CS0106](#). For more information, see [interface \(C# Reference\)](#).

Beginning with [C# 8.0](#), you can define an implementation for members declared in an interface. If a class inherits a method implementation from an interface, that method is only accessible through a reference of the interface type. The inherited member doesn't appear as part of the public interface. The following sample defines a default implementation for an interface method:

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

The following sample invokes the default implementation:

```
var sample = new SampleClass();
//sample.Paint(); // "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Any class that implements the `IControl` interface can override the default `Paint` method, either as a public method, or as an explicit interface implementation.

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Interfaces](#)
- [Inheritance](#)

How to explicitly implement interface members (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example declares an `interface`, `IDimensions`, and a class, `Box`, which explicitly implements the interface members `GetLength` and `GetWidth`. The members are accessed through the interface instance `dimensions`.

Example

```

interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
   Length: 30
   Width: 20
*/

```

Robust Programming

- Notice that the following lines, in the `Main` method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a `class` instance:

```

//System.Console.WriteLine("Length: {0}", box1.GetLength());
//System.Console.WriteLine("Width: {0}", box1.GetWidth());

```

- Notice also that the following lines, in the `Main` method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());  
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Interfaces](#)
- [How to explicitly implement members of two interfaces](#)

How to explicitly implement members of two interfaces (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Explicit [interface](#) implementation also allows the programmer to implement two interfaces that have the same member names and give each interface member a separate implementation. This example displays the dimensions of a box in both metric and English units. The `Box` [class](#) implements two interfaces `IEnglishDimensions` and `IMetricDimensions`, which represent the different measurement systems. Both interfaces have identical member names, `Length` and `Width`.

Example

```

// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
   Length(in): 30
   Width (in): 20
   Length(cm): 76.2
   Width (cm): 50.8
*/

```

Robust Programming

If you want to make the default measurements in English units, implement the methods Length and Width normally, and explicitly implement the Length and Width methods from the IMetricDimensions interface:

```
// Normal implementation:  
public float Length() => lengthInches;  
public float Width() => widthInches;  
  
// Explicit implementation:  
float IMetricDimensions.Length() => lengthInches * 2.54f;  
float IMetricDimensions.Width() => widthInches * 2.54f;
```

In this case, you can access the English units from the class instance and access the metric units from the interface instance:

```
public static void Test()  
{  
    Box box1 = new Box(30.0f, 20.0f);  
    IMetricDimensions mDimensions = box1;  
  
    System.Console.WriteLine("Length(in): {0}", box1.Length());  
    System.Console.WriteLine("Width (in): {0}", box1.Width());  
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());  
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());  
}
```

See also

- [C# Programming Guide](#)
- [Classes and Structs](#)
- [Interfaces](#)
- [How to explicitly implement interface members](#)

Delegates (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

A [delegate](#) is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

```
public delegate int PerformCalculation(int x, int y);
```

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.

NOTE

In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. You can write a method that compares two objects in your application. That method can be used in a delegate for a sort algorithm. Because the comparison code is separate from the library, the sort method can be more general.

[Function pointers](#) were added to C# 9 for similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked using a virtual method added to a delegate type. Using function pointers, you can specify different conventions.

Delegates Overview

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods don't have to match the delegate type exactly. For more information, see [Using Variance in Delegates](#).
- Lambda expressions are a more concise way of writing inline code blocks. Lambda expressions (in certain contexts) are compiled to delegate types. For more information about lambda expressions, see [Lambda expressions](#).

In This Section

- [Using Delegates](#)

- [When to Use Delegates Instead of Interfaces \(C# Programming Guide\)](#)
- [Delegates with Named vs. Anonymous Methods](#)
- [Using Variance in Delegates](#)
- [How to combine delegates \(Multicast Delegates\)](#)
- [How to declare, instantiate, and use a delegate](#)

C# Language Specification

For more information, see [Delegates](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

- [Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers](#)
- [Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0](#)

See also

- [Delegate](#)
- [C# Programming Guide](#)
- [Events](#)

Using Delegates (C# Programming Guide)

11/2/2020 • 5 minutes to read • [Edit Online](#)

A [delegate](#) is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate. The following example declares a delegate named `Del` that can encapsulate a method that takes a [string](#) as an argument and returns [void](#):

```
public delegate void Del(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an [anonymous function](#). Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

```
// Create a method for a delegate.  
public static void DelegateMethod(string message)  
{  
    Console.WriteLine(message);  
}
```

```
// Instantiate the delegate.  
Del handler = DelegateMethod;  
  
// Call the delegate.  
handler("Hello World");
```

Delegate types are derived from the [Delegate](#) class in .NET. Delegate types are [sealed](#)—they cannot be derived from—and it is not possible to derive custom classes from [Delegate](#). Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the `Del` type as a parameter:

```
public static void MethodWithCallback(int param1, int param2, Del callback)  
{  
    callback("The number is: " + (param1 + param2).ToString());  
}
```

You can then pass the delegate created above to that method:

```
MethodWithCallback(1, 2, handler);
```

and receive the following output to the console:

```
The number is: 3
```

Using the delegate as an abstraction, `MethodWithCallback` does not need to call the console directly—it does not have to be designed with a console in mind. What `MethodWithCallback` does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Along with the static `DelegateMethod` shown previously, we now have three methods that can be wrapped by a `Del` instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

At this point `allMethodsDelegate` contains three methods in its invocation list—`Method1`, `Method2`, and `DelegateMethod`. The original three delegates, `d1`, `d2`, and `d3`, remain unchanged. When `allMethodsDelegate` is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the [subtraction or subtraction assignment operators](#) (`-` or `-=`). For example:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Because delegate types are derived from `System.Delegate`, the methods and properties defined by that class can

be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from [MulticastDelegate](#), which is a subclass of [System.Delegate](#). The above code works in either case because both classes support [GetInvocationList](#).

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see [Events](#).

Comparing delegates of two different types assigned at compile-time will result in a compilation error. If the delegate instances are statically of the type [System.Delegate](#), then the comparison is allowed, but will return false at run time. For example:

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

See also

- [C# Programming Guide](#)
- [Delegates](#)
- [Using Variance in Delegates](#)
- [Variance in Delegates](#)
- [Using Variance for Func and Action Generic Delegates](#)
- [Events](#)

Delegates with Named vs. Anonymous Methods (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A [delegate](#) can be associated with a named method. When you instantiate a delegate by using a named method, the method is passed as a parameter, for example:

```
// Declare a delegate.  
delegate void Del(int x);  
  
// Define a named method.  
void DoWork(int k) { /* ... */ }  
  
// Instantiate the delegate using the method as a parameter.  
Del d = obj.DoWork;
```

This is called using a named method. Delegates constructed with a named method can encapsulate either a [static](#) method or an instance method. Named methods are the only way to instantiate a delegate in earlier versions of C#. However, in a situation where creating a new method is unwanted overhead, C# enables you to instantiate a delegate and immediately specify a code block that the delegate will process when it is called. The block can contain either a lambda expression or an anonymous method. For more information, see [Anonymous Functions](#).

Remarks

The method that you pass as a delegate parameter must have the same signature as the delegate declaration.

A delegate instance may encapsulate either static or instance method.

Although the delegate can use an [out](#) parameter, we do not recommend its use with multicast event delegates because you cannot know which delegate will be called.

Example 1

The following is a simple example of declaring and using a delegate. Notice that both the delegate, `Del`, and the associated method, `MultiplyNumbers`, have the same signature

```

// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
   Invoking the delegate using 'MultiplyNumbers':
   2 4 6 8 10
*/

```

Example 2

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
   A message from the instance method.
   A message from the static method.
*/
```

See also

- [C# Programming Guide](#)
- [Delegates](#)
- [How to combine delegates \(Multicast Delegates\)](#)
- [Events](#)

How to combine delegates (Multicast Delegates) (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example demonstrates how to create multicast delegates. A useful property of `delegate` objects is that multiple objects can be assigned to one delegate instance by using the `+` operator. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

The `-` operator can be used to remove a component delegate from a multicast delegate.

Example

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}

/* Output:
Invoking delegate hiDel:
Hello, A!
Invoking delegate byeDel:
Goodbye, B!
Invoking delegate multiDel:
Hello, C!
Goodbye, C!
Invoking delegate multiMinusHiDel:
Goodbye, D!
*/
```

See also

- [MulticastDelegate](#)
- [C# Programming Guide](#)
- [Events](#)

How to declare, instantiate, and use a Delegate (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

In C# 1.0 and later, delegates can be declared as shown in the following example.

```
// Declare a delegate.  
delegate void Del(string str);  
  
// Declare a method with the same signature as the delegate.  
static void Notify(string name)  
{  
    Console.WriteLine($"Notification received for: {name}");  
}
```

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

C# 2.0 provides a simpler way to write the previous declaration, as shown in the following example.

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

In C# 2.0 and later, it is also possible to use an anonymous method to declare and initialize a [delegate](#), as shown in the following example.

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
    { Console.WriteLine($"Notification received for: {name}"); };
```

In C# 3.0 and later, delegates can also be declared and instantiated by using a lambda expression, as shown in the following example.

```
// Instantiate Del by using a lambda expression.  
Del del4 = name => { Console.WriteLine($"Notification received for: {name}"); };
```

For more information, see [Lambda Expressions](#).

The following example illustrates declaring, instantiating, and using a delegate. The `BookDB` class encapsulates a bookstore database that maintains a database of books. It exposes a method, `ProcessPaperbackBooks`, which finds all paperback books in the database and calls a delegate for each one. The `delegate` type that is used is named `ProcessBookCallback`. The `Test` class uses this class to print the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is performed on the paperback books after it finds them.

Example

```
// A set of classes for handling a bookstore:  
namespace Bookstore  
{  
    using System.Collections;  
  
    // Describes a book in the book list:  
    public struct Book  
    {  
        public string Title;          // Title of the book.  
        public string Author;         // Author of the book.  
        public decimal Price;         // Price of the book.  
        public bool Paperback;        // Is it paperback?  
  
        public Book(string title, string author, decimal price, bool paperBack)  
        {  
            Title = title;  
            Author = author;  
            Price = price;  
            Paperback = paperBack;  
        }  
    }  
  
    // Declare a delegate type for processing a book:  
    public delegate void ProcessBookCallback(Book book);  
  
    // Maintains a book database.  
    public class BookDB  
    {  
        // List of all books in the database:  
        ArrayList list = new ArrayList();  
  
        // Add a book to the database:  
        public void AddBook(string title, string author, decimal price, bool paperBack)  
        {  
            list.Add(new Book(title, author, price, paperBack));  
        }  
  
        // Call a passed-in delegate on each paperback book to process it:  
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)  
        {  
            foreach (Book b in list)  
            {  
                if (b.Paperback)  
                    // Calling the delegate:  
                    processBook(b);  
            }  
        }  
    }  
  
    // Using the Bookstore classes:  
    namespace BookTestClient  
    {  
        using Bookstore;  
  
        // Class to total and average prices of books:  
        class PriceTotaller  
        {  
            int countBooks = 0;  
            decimal priceBooks = 0.0m;  
  
            internal void AddBookToTotal(Book book)  
            {  
                countBooks += 1;  
                priceBooks += book.Price;  
            }  
        }  
    }  
}
```

```

        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"    {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                totaller.AveragePrice());
        }
    }

    // Initialize the book database with some test books:
    static void AddBooks(BookDB bookDB)
    {
        bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
        bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
        bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
        bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
    }
}

/* Output:
Paperback Book Titles:
The C Programming Language
The Unicode Standard 2.0
Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/

```

Robust Programming

- Declaring a delegate.

The following statement declares a new delegate type.

```
public delegate void ProcessBookCallback(Book book);
```

Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

- Instantiating a delegate.

After a delegate type has been declared, a delegate object must be created and associated with a particular method. In the previous example, you do this by passing the `PrintTitle` method to the `ProcessPaperbackBooks` method as in the following example:

```
bookDB.ProcessPaperbackBooks(PrintTitle);
```

This creates a new delegate object associated with the `static` method `Test.PrintTitle`. Similarly, the non-static method `AddBookToTotal` on the object `totaller` is passed as in the following example:

```
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

In both cases a new delegate object is passed to the `ProcessPaperbackBooks` method.

After a delegate is created, the method it is associated with never changes; delegate objects are immutable.

- Calling a delegate.

After a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. Following is an example of a delegate call:

```
processBook(b);
```

A delegate can be either called synchronously, as in this example, or asynchronously by using `BeginInvoke` and `EndInvoke` methods.

See also

- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)

Arrays (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify `object` as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from [Object](#).

```
type[] arrayName;
```

Example

The following example creates single-dimensional, multidimensional, and jagged arrays:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Array overview

An array has the following properties:

- An array can be [single-dimensional](#), [multidimensional](#) or [jagged](#).
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to `null`.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.

- Array types are [reference types](#) derived from the abstract base type [Array](#). Since this type implements [IEnumerable](#) and [IEnumerable<T>](#), you can use [foreach](#) iteration on all arrays in C#.

Arrays as Objects

In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++.

[Array](#) is the abstract base type of all array types. You can use the properties and other class members that [Array](#) has. An example of this is using the [Length](#) property to get the length of an array. The following code assigns the length of the `numbers` array, which is `5`, to a variable called `lengthOfNumbers`:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

The [Array](#) class provides many other useful methods and properties for sorting, searching, and copying arrays.

The following example uses the [Rank](#) property to display the number of dimensions of an array.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

See also

- [How to use single-dimensional arrays](#)
- [How to use multi-dimensional arrays](#)
- [How to use jagged arrays](#)
- [Using foreach with arrays](#)
- [Passing arrays as arguments](#)
- [Implicitly typed arrays](#)
- [C# Programming Guide](#)
- [Collections](#)

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Single-Dimensional Arrays (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You create a single-dimensional array using the `new` operator specifying the array element type and the number of elements. The following example declares an array of five integers:

```
int[] array = new int[5];
```

This array contains the elements from `array[0]` to `array[4]`. The elements of the array are initialized to the default value of the element type, `0` for integers.

Arrays can store any element type you specify, such as the following example that declares an array of strings:

```
string[] stringArray = new string[6];
```

Array Initialization

You can initialize the elements of an array when you declare the array. The length specifier isn't needed because it's inferred by the number of elements in the initialization list. For example:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

The following code shows a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

You can avoid the `new` expression and the array type when you initialize an array upon declaration, as shown in the following code. This is called an [implicitly typed array](#):

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

You can declare an array variable without creating it, but you must use the `new` operator when you assign a new array to this variable. For example:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK
//array3 = {1, 3, 5, 7, 9}; // Error
```

Value Type and Reference Type Arrays

Consider the following array declaration:

```
SomeType[] array4 = new SomeType[10];
```

The result of this statement depends on whether `SomeType` is a value type or a reference type. If it's a value type, the statement creates an array of 10 elements, each of which has the type `SomeType`. If `SomeType` is a reference type, the statement creates an array of 10 elements, each of which is initialized to a null reference. In both instances, the elements are initialized to the default value for the element type. For more information about value types and reference types, see [Value types](#) and [Reference types](#).

See also

- [Array](#)
- [Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Multidimensional Arrays (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns.

```
int[,] array = new int[4, 2];
```

The following declaration creates an array of three dimensions, 4, 2, and 3.

```
int[,,] array1 = new int[4, 2, 3];
```

Array Initialization

You can initialize the array upon declaration, as is shown in the following example.

```

// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                         { "five", "six" } };

// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                         { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12

```

You can also initialize the array without specifying the rank.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

If you choose to declare an array variable without initialization, you must use the `new` operator to assign an array to the variable. The use of `new` is shown in the following example.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}}; // Error
```

The following example assigns a value to a particular array element.

```
array5[2, 1] = 25;
```

Similarly, the following example gets the value of a particular array element and assigns it to variable

```
elementValue .
```

```
int elementValue = array5[2, 1];
```

The following code example initializes the array elements to default values (except for jagged arrays).

```
int[,] array6 = new int[10, 10];
```

See also

- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Jagged Arrays](#)

Jagged Arrays (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use `jaggedArray`, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

You can also initialize the array upon declaration like this:

```
int[][] jaggedArray2 = new int[][] [
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

You can use the following shorthand form. Notice that you cannot omit the `new` operator from the elements initialization because there is no default initialization for the elements:

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.

You can access individual array elements like these examples:

```
// Assign 77 to the second element ([1]) of the first array ([0]):  
jaggedArray3[0][1] = 77;  
  
// Assign 88 to the second element ([1]) of the third array ([2]):  
jaggedArray3[2][1] = 88;
```

It's possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes. For more information, see [Multidimensional Arrays](#).

```
int[][] jaggedArray4 = new int[3][]  
{  
    new int[,] { {1,3}, {5,7} },  
    new int[,] { {0,2}, {4,6}, {8,10} },  
    new int[,] { {11,22}, {99,88}, {0,9} }  
};
```

You can access individual elements as shown in this example, which displays the value of the element `[1,0]` of the first array (value `5`):

```
System.Console.WriteLine("{0}", jaggedArray4[0][1, 0]);
```

The method `Length` returns the number of arrays contained in the jagged array. For example, assuming you have declared the previous array, this line:

```
System.Console.WriteLine(jaggedArray4.Length);
```

returns a value of 3.

Example

This example builds an array whose elements are themselves arrays. Each one of the array elements has a different size.

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Element(0): 1 3 5 7 9
   Element(1): 2 4 6 8
*/
```

See also

- [Array](#)
- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)

Using foreach with arrays (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `foreach` statement provides a simple, clean way to iterate through the elements of an array.

For single-dimensional arrays, the `foreach` statement processes elements in increasing index order, starting with index 0 and ending with index `Length - 1`:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.WriteLine("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.WriteLine("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

However, with multidimensional arrays, using a nested `for` loop gives you more control over the order in which to process the array elements.

See also

- [Array](#)
- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Passing arrays as arguments (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Arrays can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

Passing single-dimensional arrays as arguments

You can pass an initialized single-dimensional array to a method. For example, the following statement sends an array to a print method.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

The following code shows a partial implementation of the print method.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Example

In the following example, an array of strings is initialized and passed as an argument to a `DisplayArray` method for strings. The method displays the elements of the array. Next, the `ChangeArray` method reverses the array elements, and then the `ChangeArrayElements` method modifies the first three elements of the array. After each method returns, the `DisplayArray` method shows that passing an array by value doesn't prevent changes to the array elements.

```

using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}

// The example displays the following output:
//      Sun Mon Tue Wed Thu Fri Sat
//
//      Array weekDays after the call to ChangeArray:
//      Sat Fri Thu Wed Tue Mon Sun
//
//      Array weekDays after the call to ChangeArrayElements:
//      Mon Wed Fri Wed Tue Mon Sun

```

Passing multidimensional arrays as arguments

You pass an initialized multidimensional array to a method in the same way that you pass a one-dimensional array.

```

int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);

```

The following code shows a partial declaration of a print method that accepts a two-dimensional array as its argument.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example:

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Example

In the following example, a two-dimensional array of integers is initialized and passed to the `Print2DArray` method. The method displays the elements of the array.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
Element(0,0)=1
Element(0,1)=2
Element(1,0)=3
Element(1,1)=4
Element(2,0)=5
Element(2,1)=6
Element(3,0)=7
Element(3,1)=8
*/
```

See also

- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Implicitly Typed Arrays (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly-typed variable also apply to implicitly-typed arrays. For more information, see [Implicitly Typed Local Variables](#).

Implicitly-typed arrays are usually used in query expressions together with anonymous types and object and collection initializers.

The following examples show how to create an implicitly-typed array:

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

In the previous example, notice that with implicitly-typed arrays, no square brackets are used on the left side of the initialization statement. Note also that jagged arrays are initialized by using `new []` just like single-dimension arrays.

Implicitly-typed Arrays in Object Initializers

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, `contacts` is an implicitly-typed array of anonymous types, each of which contains an array named `PhoneNumbers`. Note that the `var` keyword is not used inside the object initializers.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

See also

- [C# Programming Guide](#)
- [Implicitly Typed Local Variables](#)
- [Arrays](#)
- [Anonymous Types](#)
- [Object and Collection Initializers](#)
- [var](#)
- [LINQ in C#](#)

Strings (C# Programming Guide)

11/2/2020 • 12 minutes to read • [Edit Online](#)

A string is an object of type [String](#) whose value is text. Internally, the text is stored as a sequential read-only collection of [Char](#) objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The [Length](#) property of a string represents the number of [Char](#) objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the [StringInfo](#) object.

string vs. System.String

In C#, the `string` keyword is an alias for [String](#). Therefore, `String` and `string` are equivalent, and you can use whichever naming convention you prefer. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see [string](#). For more information about the type and its methods, see [String](#).

Declaring and Initializing Strings

You can declare and initialize strings in various ways, as shown in the following example:

```
// Declare without initializing.  
string message1;  
  
// Initialize to null.  
string message2 = null;  
  
// Initialize as an empty string.  
// Use the Empty constant instead of the literal "".  
string message3 = System.String.Empty;  
  
// Initialize with a regular string literal.  
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";  
  
// Initialize with a verbatim string literal.  
string newPath = @"c:\\Program Files\\Microsoft Visual Studio 9.0";  
  
// Use System.String if you prefer.  
System.String greeting = "Hello World!";  
  
// In local variables (i.e. within a method body)  
// you can use implicit typing.  
var temp = "I'm still a strongly-typed System.String!";  
  
// Use a const string to prevent 'message4' from  
// being used to store another string value.  
const string message4 = "You can't get rid of me!";  
  
// Use the String constructor only when creating  
// a string from a char*, char[], or sbyte*. See  
// System.String documentation for details.  
char[] letters = { 'A', 'B', 'C' };  
string alphabet = new string(letters);
```

Note that you do not use the `new` operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the [Empty](#) constant value to create a new [String](#) object whose string is of zero length. The string literal representation of a zero-length string is `""`. By initializing strings with the [Empty](#) value instead of [null](#), you can reduce the chances of a [NullReferenceException](#) occurring. Use the static [IsNullOrEmpty\(String\)](#) method to verify the value of a string before you try to access it.

Immutability of String Objects

String objects are *immutable*: they cannot be changed after they have been created. All of the [String](#) methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object that was assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see [How to modify string contents](#).

Regular and Verbatim String Literals

Use regular string literals when you must embed escape characters provided by C#, as shown in the following example:

```

string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1      Column 2      Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
   Row 1
   Row 2
   Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The A\u00C6olian Harp", by Samuel Taylor Coleridge

```

Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

```

string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,....
*/

string quote = @"Her name was ""Sara."";
//Output: Her name was "Sara."

```

String Escape Sequences

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

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\v	Vertical tab	0x000B
\u	Unicode escape sequence (UTF-16)	\uHHHH (range: 0000 - FFFF; example: \u00E7 = "ç")
\U	Unicode escape sequence (UTF-32)	\U00HHHHHHH (range: 000000 - 10FFFF; example: \U0001F47D = "⌚")
\x	Unicode escape sequence similar to "\u" except with variable length	\xH[H][H][H] (range: 0 - FFFF; example: \x00E7 or \x0E7 or \xE7 = "ç")

WARNING

When using the `\x` escape sequence and specifying less than 4 hex digits, if the characters that immediately follow the escape sequence are valid hex digits (i.e. 0-9, A-F, and a-f), they will be interpreted as being part of the escape sequence. For example, `\xA1` produces "í", which is code point U+00A1. However, if the next character is "A" or "a", then the escape sequence will instead be interpreted as being `\xA1A` and produce "▀", which is code point U+0A1A. In such cases, specifying all 4 hex digits (e.g. `\x00A1`) will prevent any possible misinterpretation.

NOTE

At compile time, verbatim strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim version from your source code. For example, the verbatim string `@"C:\files.txt"` will appear in the watch window as "C:\\files.txt".

Format Strings

A format string is a string whose contents are determined dynamically at runtime. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) will be resolved to a value and output as a formatted string at runtime. There are two methods to create format strings: string interpolation and composite formatting.

String Interpolation

Available in C# 6.0 and later, *interpolated strings* are identified by the `$` special character and include interpolated expressions in braces. If you are new to string interpolation, see the [String interpolation - C# interactive tutorial](#) for a quick overview.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but improves ease of use and inline clarity.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Composite Formatting

The [String.Format](#) utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published -
pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information on formatting .NET types see [Formatting Types in .NET](#).

Substrings

A substring is any sequence of characters that is contained in a string. Use the [Substring](#) method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the [IndexOf](#) method. Use the [Replace](#) method to replace all occurrences of a specified substring with a new string. Like the [Substring](#) method, [Replace](#) actually returns a new string and does not modify the original string. For more information, see [How to search strings](#) and [How to modify string contents](#).

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing Individual Characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

If the [String](#) methods do not provide the functionality that you must have to modify individual characters in a string, you can use a [StringBuilder](#) object to modify the individual chars "in-place", and then create a new string to store the results by using the [StringBuilder](#) methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

```
string question = "hOW DOES mICROSOFT WORD DEAL WITH THE cAPS LOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null Strings and Empty Strings

An empty string is an instance of a [System.String](#) object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid [System.String](#) objects. Empty strings are initialized as follows:

```
string s = String.Empty;
```

By contrast, a null string does not refer to an instance of a [System.String](#) object and any attempt to call a method on a null string causes a [NullReferenceException](#). However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```

static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}

```

Using StringBuilder for Fast String Creation

String operations in .NET are highly optimized and in most cases do not significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The [StringBuilder](#) class creates a string buffer that offers better performance if your program performs many string manipulations. The [StringBuilder](#) string also enables you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```

System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet

```

In this example, a [StringBuilder](#) object is used to create a string from a set of numeric types:

```

using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            var sb = new StringBuilder();

            // Create a string composed of numbers 0 - 9
            for (int i = 0; i < 10; i++)
            {
                sb.Append(i.ToString());
            }
            Console.WriteLine(sb); // displays 0123456789

            // Copy one character of the string (not possible with a System.String)
            sb[0] = sb[9];

            Console.WriteLine(sb); // displays 9123456789
            Console.WriteLine();
        }
    }
}

```

Strings, Extension Methods and LINQ

Because the [String](#) type implements [IEnumerable<T>](#), you can use the extension methods defined in the [Enumerable](#) class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the [String](#) type, but they are available nevertheless. You can also use LINQ query expressions on strings. For more information, see [LINQ and Strings](#).

Related Topics

TOPIC	DESCRIPTION
How to modify string contents	Illustrates techniques to transform strings and modify the contents of strings.
How to compare strings	Shows how to perform ordinal and culture specific comparisons of strings.
How to concatenate multiple strings	Demonstrates various ways to join multiple strings into one.
How to parse strings using String.Split	Contains code examples that illustrate how to use the <code>String.Split</code> method to parse strings.
How to search strings	Explains how to use search for specific text or patterns in strings.
How to determine whether a string represents a numeric value	Shows how to safely parse a string to see whether it has a valid numeric value.
String interpolation	Describes the string interpolation feature that provides a convenient syntax to format strings.

TOPIC	DESCRIPTION
Basic String Operations	Provides links to topics that use System.String and System.Text.StringBuilder methods to perform basic string operations.
Parsing Strings	Describes how to convert string representations of .NET base types to instances of the corresponding types.
Parsing Date and Time Strings in .NET	Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
Comparing Strings	Includes information about how to compare strings and provides examples in C# and Visual Basic.
Using the StringBuilder Class	Describes how to create and modify dynamic string objects by using the StringBuilder class.
LINQ and Strings	Provides information about how to perform various string operations by using LINQ queries.
C# Programming Guide	Provides links to topics that explain programming constructs in C#.

How to determine whether a string represents a numeric value (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

To determine whether a string is a valid representation of a specified numeric type, use the static `TryParse` method that is implemented by all primitive numeric types and also by types such as `DateTime` and `IPAddress`. The following example shows how to determine whether "108" is a valid `int`.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

If the string contains nonnumeric characters or the numeric value is too large or too small for the particular type you have specified, `TryParse` returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.

NOTE

A string may contain only numeric characters and still not be valid for the type whose `TryParse` method that you use. For example, "256" is not a valid value for `byte` but it is valid for `int`. "98.6" is not a valid value for `int` but it is a valid `decimal`.

Example

The following examples show how to use `TryParse` with string representations of `long`, `byte`, and `decimal` values.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
    Console.WriteLine("number1 now = {0}", number1);
else
    Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
    Console.WriteLine("number2 now = {0}", number2);
else
    Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
    Console.WriteLine("number3 now = {0}", number3);
else
    Console.WriteLine("number3 is not a valid decimal");
```

Robust Programming

Primitive numeric types also implement the `Parse` static method, which throws an exception if the string is not a valid number. `TryParse` is generally more efficient because it just returns false if the number is not valid.

.NET Security

Always use the `TryParse` or `Parse` methods to validate user input from controls such as text boxes and combo boxes.

See also

- [How to convert a byte array to an int](#)
- [How to convert a string to a number](#)
- [How to convert between hexadecimal strings and numeric types](#)
- [Parsing Numeric Strings](#)
- [Formatting Types](#)

Indexers (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble [properties](#) except that their accessors take parameters.

The following example defines a generic class with simple [get](#) and [set](#) accessor methods to assign and retrieve values. The `Program` class creates an instance of this class for storing strings.

```
using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get { return arr[i]; }
        set { arr[i] = value; }
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World";
        Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.
```

NOTE

For more examples, see [Related Sections](#).

Expression Body Definitions

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. Starting with C# 6, a read-only indexer can be implemented as an expression-bodied member, as the following example shows.

```

using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];
    int nextIndex = 0;

    // Define the indexer to allow client code to use [] notation.
    public T this[int i] => arr[i];

    public void Add(T value)
    {
        if (nextIndex >= arr.Length)
            throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
        arr[nextIndex++] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection.Add("Hello, World");
        System.Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.

```

Note that `=>` introduces the expression body, and that the `get` keyword is not used.

Starting with C# 7.0, both the get and set accessor can be implemented as expression-bodied members. In this case, both `get` and `set` keywords must be used. For example:

```

using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get => arr[i];
        set => arr[i] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World.";
        Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.

```

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The `this` keyword is used to define the indexer.
- The `value` keyword is used to define the value being assigned by the `set` accessor.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Related Sections

- [Using Indexers](#)
- [Indexers in Interfaces](#)
- [Comparison Between Properties and Indexers](#)
- [Restricting Accessor Accessibility](#)

C# Language Specification

For more information, see [Indexers](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Properties](#)

Using indexers (C# Programming Guide)

11/2/2020 • 6 minutes to read • [Edit Online](#)

Indexers are a syntactic convenience that enable you to create a [class](#), [struct](#), or [interface](#) that client applications can access as an array. The compiler will generate an `Item` property (or an alternatively named property if [IndexerNameAttribute](#) is present), and the appropriate accessor methods. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class `TempRecord` that represents the temperature in Fahrenheit as recorded at 10 different times during a 24-hour period. The class contains a `temps` array of type `float[]` to store the temperature values. By implementing an indexer in this class, clients can access the temperatures in a `TempRecord` instance as `float temp = tempRecord[4]` instead of as `float temp = tempRecord.temps[4]`. The indexer notation not only simplifies the syntax for client applications; it also makes the class, and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the `this` keyword, as the following example shows:

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

IMPORTANT

Declaring an indexer will automatically generate a property named `Item` on the object. The `Item` property is not directly accessible from the instance [member access expression](#). Additionally, if you add your own `Item` property to an object with an indexer, you'll get a [CS0102 compiler error](#). To avoid this error, use the [IndexerNameAttribute](#) rename the indexer as detailed below.

Remarks

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see [Access Modifiers](#).

For more information about how to use indexers with an interface, see [Interface Indexers](#).

The signature of an indexer consists of the number and types of its formal parameters. It doesn't include the indexer type or the names of the formal parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, you cannot pass an indexer value as a `ref` or `out` parameter.

To provide the indexer with a name that other languages can use, use [System.Runtime.CompilerServices.IndexerNameAttribute](#), as the following example shows:

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

This indexer will have the name `TheItem`, as it is overridden by the indexer name attribute. By default, the indexer name is `Item`.

Example 1

The following example shows how to declare a private array field, `temps`, and an indexer. The indexer enables direct access to the instance `tempRecord[i]`. The alternative to using the indexer is to declare the array as a `public` member and access its members, `tempRecord.temps[i]`, directly.

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Notice that when an indexer's access is evaluated, for example, in a `Console.WriteLine` statement, the `get` accessor is invoked. Therefore, if no `get` accessor exists, a compile-time error occurs.

```

using System;

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
     * Element #0 = 56.2
     * Element #1 = 56.7
     * Element #2 = 56.5
     * Element #3 = 58.3
     * Element #4 = 58.8
     * Element #5 = 60.1
     * Element #6 = 65.9
     * Element #7 = 62.1
     * Element #8 = 59.2
     * Element #9 = 57.5
    */
}

```

Indexing using other values

C# doesn't limit the indexer parameter type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can coexist.

Example 2

The following example declares a class that stores the days of the week. A `get` accessor takes a string, the name of a day, and returns the corresponding integer. For example, "Sunday" returns 0, "Monday" returns 1, and so on.

```

using System;

// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form \"Sun\", \"Mon\", etc");
    }
}

```

Consuming example 2

```

using System;

class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week["Fri"]);

        try
        {
            Console.WriteLine(week["Made-up day"]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day Made-up day is not supported.
    // Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
}

```

Example 3

The following example declares a class that stores the days of the week using the [System.DayOfWeek](#) enum. A `get` accessor takes a `DayOfWeek`, the value of a day, and returns the corresponding integer. For example, `DayOfWeek.Sunday` returns 0, `DayOfWeek.Monday` returns 1, and so on.

```

using System;
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    {
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined System.DayOfWeek value.");
    }
}

```

Consuming example 3

```

using System;

class Program
{
    static void Main()
    {
        var week = new DayOfWeekCollection();
        Console.WriteLine(week[DayOfWeek.Friday]);

        try
        {
            Console.WriteLine(week[(DayOfWeek)43]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}

```

Robust programming

There are two main ways in which the security and reliability of indexers can be improved:

- Be sure to incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value. In the first example earlier in this topic, the TempRecord class provides a Length property that enables the client code to verify the input before passing it to the indexer. You can also put the error handling code inside the indexer itself. Be sure to document for users any exceptions that you

throw inside an indexer accessor.

- Set the accessibility of the `get` and `set` accessors to be as restrictive as is reasonable. This is important for the `set` accessor in particular. For more information, see [Restricting Accessor Accessibility](#).

See also

- [C# Programming Guide](#)
- [Indexers](#)
- [Properties](#)

Indexers in Interfaces (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Indexers can be declared on an [interface](#). Accessors of interface indexers differ from the accessors of [class](#) indexers in the following ways:

- Interface accessors do not use modifiers.
- An interface accessor typically does not have a body.

The purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only. You may provide an implementation for an indexer defined in an interface, but this is rare. Indexers typically define an API to access data fields, and data fields cannot be defined in an interface.

The following is an example of an interface indexer accessor:

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]    // indexer declaration
    {
        // The arr object will throw IndexOutOfRangeException.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```

IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
   Element #0 = 360877544
   Element #1 = 327058047
   Element #2 = 1913480832
   Element #3 = 1519039937
   Element #4 = 601472233
   Element #5 = 323352310
   Element #6 = 1422639981
   Element #7 = 1797892494
   Element #8 = 875761049
   Element #9 = 393083859
*/

```

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example

```

string IIndexInterface.this[int index]
{
}

```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if an `Employee` class is implementing two interfaces, `ICitizen` and `IEmployee`, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

```

string IEmployee.this[int index]
{
}

```

implements the indexer on the `IEmployee` interface, while the following declaration:

```

string ICitizen.this[int index]
{
}

```

implements the indexer on the `ICitizen` interface.

See also

- [C# Programming Guide](#)
- [Indexers](#)
- [Properties](#)
- [Interfaces](#)

Comparison Between Properties and Indexers (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Indexers are like properties. Except for the differences shown in the following table, all the rules that are defined for property accessors apply to indexer accessors also.

PROPERTY	INDEXER
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A <code>get</code> accessor of a property has no parameters.	A <code>get</code> accessor of an indexer has the same formal parameter list as the indexer.
A <code>set</code> accessor of a property contains the implicit <code>value</code> parameter.	A <code>set</code> accessor of an indexer has the same formal parameter list as the indexer, and also to the <code>value</code> parameter.
Supports shortened syntax with Auto-Implemented Properties .	Supports expression bodied members for get only indexers.

See also

- [C# Programming Guide](#)
- [Indexers](#)
- [Properties](#)

Events (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Events enable a [class](#) or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see [How to subscribe to and unsubscribe from events](#).

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [Calling Synchronous Methods Asynchronously](#).
- In the .NET class library, events are based on the [EventHandler](#) delegate and the [EventArgs](#) base class.

Related Sections

For more information, see:

- [How to subscribe to and unsubscribe from events](#)
- [How to publish events that conform to .NET Guidelines](#)
- [How to raise base class events in derived classes](#)
- [How to implement interface events](#)
- [How to implement custom event accessors](#)

C# Language Specification

For more information, see [Events](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- [EventHandler](#)
- [C# Programming Guide](#)
- [Delegates](#)
- [Creating Event Handlers in Windows Forms](#)

How to subscribe to and unsubscribe from events (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

You subscribe to an event that is published by another class when you want to write custom code that is called when that event is raised. For example, you might subscribe to a button's `click` event in order to make your application do something useful when the user clicks the button.

To subscribe to events by using the Visual Studio IDE

1. If you cannot see the **Properties** window, in **Design** view, right-click the form or control for which you want to create an event handler, and select **Properties**.
2. On top of the **Properties** window, click the **Events** icon.
3. Double-click the event that you want to create, for example the `Load` event.

Visual C# creates an empty event handler method and adds it to your code. Alternatively you can add the code manually in **Code** view. For example, the following lines of code declare an event handler method that will be called when the `Form` class raises the `Load` event.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Add your form load event handling code here.
}
```

The line of code that is required to subscribe to the event is also automatically generated in the `InitializeComponent` method in the `Form1.Designer.cs` file in your project. It resembles this:

```
this.Load += new System.EventHandler(this.Form1_Load);
```

To subscribe to events programmatically

1. Define an event handler method whose signature matches the delegate signature for the event. For example, if the event is based on the `EventHandler` delegate type, the following code represents the method stub:

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
    // Do something useful here.
}
```

2. Use the addition assignment operator (`+=`) to attach an event handler to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent`. Note that the subscriber class needs a reference to the publisher class in order to subscribe to its events.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

Note that the previous syntax is new in C# 2.0. It is exactly equivalent to the C# 1.0 syntax in which the encapsulating delegate must be explicitly created by using the `new` keyword:

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

You can also use a [lambda expression](#) to specify an event handler:

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

To subscribe to events by using an anonymous method

- If you will not have to unsubscribe to an event later, you can use the addition assignment operator (`+=`) to attach an anonymous method to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent` and that a `CustomEventArgs` class has also been defined to carry some kind of specialized event information. Note that the subscriber class needs a reference to `publisher` in order to subscribe to its events.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

It is important to notice that you cannot easily unsubscribe from an event if you used an anonymous function to subscribe to it. To unsubscribe in this scenario, it is necessary to go back to the code where you subscribe to the event, store the anonymous method in a delegate variable, and then add the delegate to the event. In general, we recommend that you do not use anonymous functions to subscribe to events if you will have to unsubscribe from the event at some later point in your code. For more information about anonymous functions, see [Anonymous Functions](#).

Unsubscribing

To prevent your event handler from being invoked when the event is raised, unsubscribe from the event. In order to prevent resource leaks, you should unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference to the delegate that encapsulates the subscriber's event handler. As long as the publishing object holds that reference, garbage collection will not delete your subscriber object.

To unsubscribe from an event

- Use the subtraction assignment operator (`-=`) to unsubscribe from an event:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to `null`.

See also

- [Events](#)
- [event](#)

- How to publish events that conform to .NET Guidelines
- - and -= operators
- + and += operators

How to publish events that conform to .NET Guidelines (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The following procedure demonstrates how to add events that follow the standard .NET pattern to your classes and structs. All events in the .NET class library are based on the [EventHandler](#) delegate, which is defined as follows:

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

NOTE

.NET Framework 2.0 introduces a generic version of this delegate, [EventHandler<TEventArgs>](#). The following examples show how to use both versions.

Although events in classes that you define can be based on any valid delegate type, even delegates that return a value, it is generally recommended that you base your events on the .NET pattern by using [EventHandler](#), as shown in the following example.

The name `EventHandler` can lead to a bit of confusion as it doesn't actually handle the event. The [EventHandler](#), and generic [EventHandler<TEventArgs>](#) are delegate types. A method or lambda expression whose signature matches the delegate definition is the *event handler* and will be invoked when the event is raised.

Publish events based on the EventHandler pattern

1. (Skip this step and go to Step 3a if you do not have to send custom data with your event.) Declare the class for your custom data at a scope that is visible to both your publisher and subscriber classes. Then add the required members to hold your custom event data. In this example, a simple string is returned.

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string message)
    {
        Message = message;
    }

    public string Message { get; set; }
}
```

2. (Skip this step if you are using the generic version of [EventHandler<TEventArgs>](#).) Declare a delegate in your publishing class. Give it a name that ends with `EventHandler`. The second parameter specifies your custom `EventArgs` type.

```
public delegate void CustomEventHandler(object sender, CustomEventArgs args);
```

3. Declare the event in your publishing class by using one of the following steps.

- a. If you have no custom `EventArgs` class, your Event type will be the non-generic `EventHandler` delegate. You do not have to declare the delegate because it is already declared in the [System](#)

namespace that is included when you create your C# project. Add the following code to your publisher class.

```
public event EventHandler RaiseCustomEvent;
```

- b. If you are using the non-generic version of [EventHandler](#) and you have a custom class derived from [EventArgs](#), declare your event inside your publishing class and use your delegate from step 2 as the type.

```
public event CustomEventHandler RaiseCustomEvent;
```

- c. If you are using the generic version, you do not need a custom delegate. Instead, in your publishing class, you specify your event type as `EventHandler<CustomEventArgs>`, substituting the name of your own class between the angle brackets.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Example

The following example demonstrates the previous steps by using a custom EventArgs class and [EventHandler<TEventArgs>](#) as the event type.

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;
```

```

// Event will be null if there are no subscribers
if (raiseEvent != null)
{
    // Format the string to send inside the CustomEventArgs parameter
    e.Message += $" at {DateTime.Now}";

    // Call to raise the event.
    raiseEvent(this, e);
}
}

//Class that subscribes to an event
class Subscriber
{
    private readonly string _id;

    public Subscriber(string id, Publisher pub)
    {
        _id = id;

        // Subscribe to the event
        pub.RaiseCustomEvent += HandleCustomEvent;
    }

    // Define what actions to take when the event is raised.
    void HandleCustomEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine($"{_id} received this message: {e.Message}");
    }
}

class Program
{
    static void Main()
    {
        var pub = new Publisher();
        var sub1 = new Subscriber("sub1", pub);
        var sub2 = new Subscriber("sub2", pub);

        // Call the method that raises the event.
        pub.DoSomething();

        // Keep the console window open
        Console.WriteLine("Press any key to continue...");
        Console.ReadLine();
    }
}
}

```

See also

- [Delegate](#)
- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)

How to raise base class events in derived classes (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The following simple example shows the standard way to declare events in a base class so that they can also be raised from derived classes. This pattern is used extensively in Windows Forms classes in the .NET class libraries.

When you create a class that can be used as a base class for other classes, you should consider the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Derived classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, most of the time, you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

NOTE

Do not declare virtual events in a base class and override them in a derived class. The C# compiler does not handle these correctly and it is unpredictable whether a subscriber to the derived event will actually be subscribing to the base class event.

Example

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
        }
    }
}
```

```

        // Safely raise the event for all subscribers
        ShapeChanged?.Invoke(this, e);
    }
}

public class Circle : Shape
{
    private double _radius;

    public Circle(double radius)
    {
        _radius = radius;
        _area = 3.14 * _radius * _radius;
    }

    public void Update(double d)
    {
        _radius = d;
        _area = 3.14 * _radius * _radius;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any circle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
{
    private double _length;
    private double _width;

    public Rectangle(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
    }

    public void Update(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any rectangle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}

```

```

}

// Represents the surface on which the shapes are drawn
// Subscribes to shape events so that it knows
// when to redraw a shape.
public class ShapeContainer
{
    private readonly List<Shape> _list;

    public ShapeContainer()
    {
        _list = new List<Shape>();
    }

    public void AddShape(Shape shape)
    {
        _list.Add(shape);

        // Subscribe to the base class event.
        shape.ShapeChanged += HandleShapeChanged;
    }

    // ...Other methods to draw, resize, etc.

    private void HandleShapeChanged(object sender, ShapeEventArgs e)
    {
        if (sender is Shape shape)
        {
            // Diagnostic message for demonstration purposes.
            Console.WriteLine($"Received event. Shape area is now {e.NewArea}");

            // Redraw the shape here.
            shape.Draw();
        }
    }
}

class Test
{
    static void Main()
    {
        //Create the event publishers and subscriber
        var circle = new Circle(54);
        var rectangle = new Rectangle(12, 9);
        var container = new ShapeContainer();

        // Add the shapes to the container.
        container.AddShape(circle);
        container.AddShape(rectangle);

        // Cause some events to be raised.
        circle.Update(57);
        rectangle.Update(7, 7);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to continue...");
        Console.ReadKey();
    }
}
/* Output:
   Received event. Shape area is now 10201.86
   Drawing a circle
   Received event. Shape area is now 49
   Drawing a rectangle
*/

```

See also

- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)
- [Access Modifiers](#)
- [Creating Event Handlers in Windows Forms](#)

How to implement interface events (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

An [interface](#) can declare an [event](#). The following example shows how to implement interface events in a class. Basically the rules are the same as when you implement any interface method or property.

To implement interface events in a class

Declare the event in your class and then invoke it in the appropriate areas.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event...

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }
}
```

Example

The following example shows how to handle the less-common situation in which your class inherits from two or more interfaces and each interface has an event with the same name. In this situation, you must provide an explicit interface implementation for at least one of the events. When you write an explicit interface implementation for an event, you must also write the `add` and `remove` event accessors. Normally these are provided by the compiler, but in this case the compiler cannot provide them.

By providing your own accessors, you can specify whether the two events are represented by the same event in your class, or by different events. For example, if the events should be raised at different times according to the interface specifications, you can associate each event with a separate implementation in your class. In the following example, subscribers determine which `OnDraw` event they will receive by casting the shape reference to either an `Ishape` or an `IDrawingObject`.

```

namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }
    }
}

```

```

        // For the sake of simplicity this one method
        // implements both interfaces.
    public void Draw()
    {
        // Raise IDrawingObject's event before the object is drawn.
        PreDrawEvent?.Invoke(this, EventArgs.Empty);

        Console.WriteLine("Drawing a shape.");

        // Raise IShape's event after the object is drawn.
        PostDrawEvent?.Invoke(this, EventArgs.Empty);
    }
}

public class Subscriber1
{
    // References the shape object as an IDrawingObject
    public Subscriber1(Shape shape)
    {
        IDrawingObject d = (IDrawingObject)shape;
        d.OnDraw += d_OnDraw;
    }

    void d_OnDraw(object sender, EventArgs e)
    {
        Console.WriteLine("Sub1 receives the IDrawingObject event.");
    }
}

// References the shape object as an IShape
public class Subscriber2
{
    public Subscriber2(Shape shape)
    {
        IShape d = (IShape)shape;
        d.OnDraw += d_OnDraw;
    }

    void d_OnDraw(object sender, EventArgs e)
    {
        Console.WriteLine("Sub2 receives the IShape event.");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Shape shape = new Shape();
        Subscriber1 sub = new Subscriber1(shape);
        Subscriber2 sub2 = new Subscriber2(shape);
        shape.Draw();

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
}

/* Output:
   Sub1 receives the IDrawingObject event.
   Drawing a shape.
   Sub2 receives the IShape event.
*/

```

See also

- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)
- [Explicit Interface Implementation](#)
- [How to raise base class events in derived classes](#)

How to implement custom event accessors (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An event is a special kind of multicast delegate that can only be invoked from within the class that it is declared in. Client code subscribes to the event by providing a reference to a method that should be invoked when the event is fired. These methods are added to the delegate's invocation list through event accessors, which resemble property accessors, except that event accessors are named `add` and `remove`. In most cases, you do not have to supply custom event accessors. When no custom event accessors are supplied in your code, the compiler will add them automatically. However, in some cases you may have to provide custom behavior. One such case is shown in the topic [How to implement interface events](#).

Example

The following example shows how to implement custom add and remove event accessors. Although you can substitute any code inside the accessors, we recommend that you lock the event before you add or remove a new event handler method.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

See also

- [Events](#)
- [event](#)

Generics (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Generics introduce the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter `T`, you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }

    class TestGenericList
    {
        private class ExampleClass { }

        static void Main()
        {
            // Declare a list of type int.
            GenericList<int> list1 = new GenericList<int>();
            list1.Add(1);

            // Declare a list of type string.
            GenericList<string> list2 = new GenericList<string>();
            list2.Add("");

            // Declare a list of type ExampleClass.
            GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
            list3.Add(new ExampleClass());
        }
    }
}
```

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. The [System.Collections.Generic](#) namespace contains several generic-based collection classes. The non-generic collections, such as [ArrayList](#) are not recommended and are maintained for compatibility purposes. For more information, see [Generics in .NET](#).

Of course, you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the `List<T>` class provided by .NET instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

Note that `T` is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

```

// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumarator<T> GetEnumarator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}

```

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Generics overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET class library contains several generic collection classes in the [System.Collections.Generic](#) namespace. These should be used whenever possible instead of classes such as [ArrayList](#) in the [System.Collections](#) namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related sections

- [Generic Type Parameters](#)
- [Constraints on Type Parameters](#)
- [Generic Classes](#)
- [Generic Interfaces](#)
- [Generic Methods](#)
- [Generic Delegates](#)
- [Differences Between C++ Templates and C# Generics](#)
- [Generics and Reflection](#)
- [Generics in the Run Time](#)

C# language specification

For more information, see the [C# Language Specification](#).

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Types](#)
- [<typeparam>](#)

- <typeparamref>
- Generics in .NET

Generic type parameters (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

In a generic type or method definition, a type parameter is a placeholder for a specific type that a client specifies when they create an instance of the generic type. A generic class, such as `GenericList<T>` listed in [Introduction to Generics](#), cannot be used as-is because it is not really a type; it is more like a blueprint for a type. To use `GenericList<T>`, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets. The type argument for this particular class can be any type recognized by the compiler. Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

In each of these instances of `GenericList<T>`, every occurrence of `T` in the class is substituted at run time with the type argument. By means of this substitution, we have created three separate type-safe and efficient objects using a single class definition. For more information on how this substitution is performed by the CLR, see [Generics in the Run Time](#).

Type parameter naming guidelines

- Do name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name would not add value.

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- Consider using `T` as the type parameter name for types with one single letter type parameter.

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- Do prefix descriptive type parameter names with "T".

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- Consider indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` may be called `TSession`.

The code analysis rule [CA1715](#) can be used to ensure that type parameters are named appropriately.

See also

- [System.Collections.Generic](#)

- [C# Programming Guide](#)
- [Generics](#)
- [Differences Between C++ Templates and C# Generics](#)

Constraints on type parameters (C# Programming Guide)

3/6/2021 • 10 minutes to read • [Edit Online](#)

Constraints inform the compiler about the capabilities a type argument must have. Without any constraints, the type argument could be any type. The compiler can only assume the members of [System.Object](#), which is the ultimate base class for any .NET type. For more information, see [Why use constraints](#). If client code uses a type that doesn't satisfy a constraint, the compiler issues an error. Constraints are specified by using the `where` contextual keyword. The following table lists the various types of constraints:

CONSTRAINT	DESCRIPTION
<code>where T : struct</code>	The type argument must be a non-nullable value type . For information about nullable value types, see Nullable value types . Because all value types have an accessible parameterless constructor, the <code>struct</code> constraint implies the <code>new()</code> constraint and can't be combined with the <code>new()</code> constraint. You can't combine the <code>struct</code> constraint with the <code>unmanaged</code> constraint.
<code>where T : class</code>	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type. In a nullable context in C# 8.0 or later, <code>T</code> must be a non-nullable reference type.
<code>where T : class?</code>	The type argument must be a reference type, either nullable or non-nullable. This constraint applies also to any class, interface, delegate, or array type.
<code>where T : notnull</code>	The type argument must be a non-nullable type. The argument can be a non-nullable reference type in C# 8.0 or later, or a non-nullable value type.
<code>where T : unmanaged</code>	The type argument must be a non-nullable unmanaged type . The <code>unmanaged</code> constraint implies the <code>struct</code> constraint and can't be combined with either the <code>struct</code> or <code>new()</code> constraints.
<code>where T : new()</code>	The type argument must have a public parameterless constructor. When used together with other constraints, the <code>new()</code> constraint must be specified last. The <code>new()</code> constraint can't be combined with the <code>struct</code> and <code>unmanaged</code> constraints.
<code>where T : <base class name></code>	The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, <code>T</code> must be a non-nullable reference type derived from the specified base class.

CONSTRAINT	DESCRIPTION
<code>where T : <base class name>?</code>	The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, <code>T</code> may be either a nullable or non-nullable type derived from the specified base class.
<code>where T : <interface name></code>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context in C# 8.0 and later, <code>T</code> must be a non-nullable type that implements the specified interface.
<code>where T : <interface name>?</code>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context in C# 8.0, <code>T</code> may be a nullable reference type, a non-nullable reference type, or a value type. <code>T</code> may not be a nullable value type.
<code>where T : U</code>	The type argument supplied for <code>T</code> must be or derive from the argument supplied for <code>U</code> . In a nullable context, if <code>U</code> is a non-nullable reference type, <code>T</code> must be non-nullable reference type. If <code>U</code> is a nullable reference type, <code>T</code> may be either nullable or non-nullable.

Why use constraints

Constraints specify the capabilities and expectations of a type parameter. Declaring those constraints means you can use the operations and method calls of the constraining type. If your generic class or method uses any operation on the generic members beyond simple assignment or calling any methods not supported by `System.Object`, you'll have to apply constraints to the type parameter. For example, the base class constraint tells the compiler that only objects of this type or derived from this type will be used as type arguments. Once the compiler has this guarantee, it can allow methods of that type to be called in the generic class. The following code example demonstrates the functionality you can add to the `GenericList<T>` class (in [Introduction to Generics](#)) by applying a base class constraint.

```

public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}

```

The constraint enables the generic class to use the `Employee.Name` property. The constraint specifies that all items of type `T` are guaranteed to be either an `Employee` object or an object that inherits from `Employee`.

Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types, as follows:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

When applying the `where T : class` constraint, avoid the `==` and `!=` operators on the type parameter because these operators will test for reference identity only, not for value equality. This behavior occurs even if these operators are overloaded in a type that is used as an argument. The following code illustrates this point; the output is false even though the `String` class overloads the `==` operator.

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

The compiler only knows that `T` is a reference type at compile time and must use the default operators that are valid for all reference types. If you must test for value equality, the recommended way is to also apply the `where T : IEquatable<T>` or `where T : IComparable<T>` constraint and implement the interface in any class that will be used to construct the generic class.

Constraining multiple parameters

You can apply constraints to multiple parameters, and multiple constraints to a single parameter, as shown in the following example:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Unbounded type parameters

Type parameters that have no constraints, such as `T` in public class `SampleClass<T>{}`, are called unbounded type parameters. Unbounded type parameters have the following rules:

- The `!=` and `==` operators can't be used because there's no guarantee that the concrete type argument will support these operators.
- They can be converted to and from `System.Object` or explicitly converted to any interface type.
- You can compare them to `null`. If an unbounded parameter is compared to `null`, the comparison will always return false if the type argument is a value type.

Type parameters as constraints

The use of a generic type parameter as a constraint is useful when a member function with its own type parameter has to constrain that parameter to the type parameter of the containing type, as shown in the following example:

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

In the previous example, `T` is a type constraint in the context of the `Add` method, and an unbounded type parameter in the context of the `List` class.

Type parameters can also be used as constraints in generic class definitions. The type parameter must be declared within the angle brackets together with any other type parameters:

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

The usefulness of type parameters as constraints with generic classes is limited because the compiler can assume nothing about the type parameter except that it derives from `System.Object`. Use type parameters as constraints on generic classes in scenarios in which you want to enforce an inheritance relationship between two type parameters.

NotNull constraint

Beginning with C# 8.0 in a nullable context, you can use the `notnull` constraint to specify that the type argument must be a non-nullable value type or non-nullable reference type. The `notnull` constraint can only be used in a `nullable enable` context. The compiler generates a warning if you add the `notnull` constraint in a nullable oblivious context.

Unlike other constraints, when a type argument violates the `notnull` constraint, the compiler generates a warning when that code is compiled in a `nullable enable` context. If the code is compiled in a nullable oblivious context, the compiler doesn't generate any warnings or errors.

Beginning with C# 8.0 in a nullable context, the `class` constraint specifies that the type argument must be a non-nullable reference type. In a nullable context, when a type parameter is a nullable reference type, the compiler generates a warning.

Unmanaged constraint

Beginning with C# 7.3, you can use the `unmanaged` constraint to specify that the type parameter must be a non-nullable [unmanaged type](#). The `unmanaged` constraint enables you to write reusable routines to work with types that can be manipulated as blocks of memory, as shown in the following example:

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

The preceding method must be compiled in an `unsafe` context because it uses the `sizeof` operator on a type not known to be a built-in type. Without the `unmanaged` constraint, the `sizeof` operator is unavailable.

The `unmanaged` constraint implies the `struct` constraint and can't be combined with it. Because the `struct`

constraint implies the `new()` constraint, the `unmanaged` constraint can't be combined with the `new()` constraint as well.

Delegate constraints

Also beginning with C# 7.3, you can use `System.Delegate` or `System.MulticastDelegate` as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. The `System.Delegate` constraint enables you to write code that works with delegates in a type-safe manner. The following code defines an extension method that combines two delegates provided they're the same type:

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source, TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

You can use the above method to combine delegates that are the same type:

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

If you uncomment the last line, it won't compile. Both `first` and `test` are delegate types, but they're different delegate types.

Enum constraints

Beginning in C# 7.3, you can also specify the `System.Enum` type as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. Generics using `System.Enum` provide type-safe programming to cache results from using the static methods in `System.Enum`. The following sample finds all the valid values for an enum type, and then builds a dictionary that maps those values to its string representation.

```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

`Enum.GetValues` and `Enum.GetName` use reflection, which has performance implications. You can call `EnumNamedValues` to build a collection that is cached and reused rather than repeating the calls that require reflection.

You could use it as shown in the following sample to create an enum and build a dictionary of its values and names:

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Generic Classes](#)
- [new Constraint](#)

Generic Classes (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see [Generic Collections in .NET](#).

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters (See [Constraints on Type Parameters](#)).

A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the `as` operator on `T`, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as `IComparable<T>` where `T` is the type of your class.

For an example of a simple generic class, see [Introduction to Generics](#).

The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class `Node<T>`, client code can reference the class either by specifying a type argument, to create a closed constructed type (`Node<int>`). Alternatively, it can leave the type parameter unspecified, for example when you specify a generic base class, to create an open constructed type (`Node<T>`). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

```

class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }

```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

```

//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}

```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as demonstrated in the following code:

```

class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}

```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

```

class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }

```

Generic types can use multiple type parameters and constraints, as follows:

```

class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }

```

Open constructed and closed constructed types can be used as method parameters:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Generics](#)
- [Saving the State of Enumerators](#)
- [An Inheritance Puzzle, Part One](#)

Generic Interfaces (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. The preference for generic classes is to use generic interfaces, such as `IComparable<T>` rather than `IComparable`, in order to avoid boxing and unboxing operations on value types. The .NET class library defines several generic interfaces for use with the collection classes in the `System.Collections.Generic` namespace.

When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used. The following code example shows a `SortedList<T>` class that derives from the `GenericList<T>` class. For more information, see [Introduction to Generics](#). `SortedList<T>` adds the constraint `where T : IComparable<T>`. This enables the `BubbleSort` method in `SortedList<T>` to use the generic `CompareTo` method on list elements. In this example, list elements are a simple class, `Person`, that implements `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data; //T as private member datatype

        public Node(T t) //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList() //constructor
    {
        head = null;
    }

    public void AddHead(T t) //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }
}
```

```

// Implementation of the iterator
public System.Collections.Generic.IEnumerator<T> GetEnumerator()
{
    Node current = head;
    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

// IEnumerable<T> inherits from IEnumerable, therefore this class
// must implement both the generic and non-generic versions of
// GetEnumerator. In most cases, the non-generic method can
// simply call the generic method.
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                // Because we need to call this method, the SortedList
                // class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

```

```
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
    }
}
```

```

        }

        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}

```

Multiple interfaces can be specified as constraints on a single type, as follows:

```

class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}

```

An interface can define more than one type parameter, as follows:

```

interface IDictionary<K, V>
{
}

```

The rules of inheritance that apply to classes also apply to interfaces:

```

interface IMonth<T> { }

interface IJanuary : IMonth<int> { } //No error
interface IFebruary<T> : IMonth<int> { } //No error
interface IMarch<T> : IMonth<T> { } //No error
//interface IApril<T> : IMonth<T, U> {} //Error

```

Generic interfaces can inherit from non-generic interfaces if the generic interface is covariant, which means it only uses its type parameter as a return value. In the .NET class library, `IEnumerable<T>` inherits from `IEnumerable` because `IEnumerable<T>` only uses `T` in the return value of `GetEnumerator` and in the `Current` property getter.

Concrete classes can implement closed constructed interfaces, as follows:

```

interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }

```

Generic classes can implement generic interfaces or closed constructed interfaces as long as the class parameter list supplies all arguments required by the interface, as follows:

```

interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { } //No error
class SampleClass2<T> : IBaseInterface2<T, string> { } //No error

```

The rules that control method overloading are the same for methods within generic classes, generic structs, or

generic interfaces. For more information, see [Generic Methods](#).

See also

- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [interface](#)
- [Generics](#)

Generic Methods (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A generic method is a method that is declared with type parameters, as follows:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

The following code example shows one way to call the method by using `int` for the type argument:

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

You can also omit the type argument and the compiler will infer it. The following call to `Swap` is equivalent to the previous call:

```
Swap(ref a, ref b);
```

The same rules for type inference apply to static methods and instance methods. The compiler can infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters only from a constraint or return value. Therefore type inference does not work with methods that have no parameters. Type inference occurs at compile time before the compiler tries to resolve overloaded method signatures. The compiler applies type inference logic to all generic methods that share the same name. In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

If you define a generic method that takes the same type parameters as the containing class, the compiler generates warning [CS0693](#) because within the method scope, the argument supplied for the inner `T` hides the argument supplied for the outer `T`. If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as shown in `GenericList2<T>` in the following example.

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

Use constraints to enable more specialized operations on type parameters in methods. This version of `Swap<T>`, now named `SwapIfGreater<T>`, can only be used with type arguments that implement `IComparable<T>`.

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Generic methods can be overloaded on several type parameters. For example, the following methods can all be located in the same class:

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

C# Language Specification

For more information, see the [C# Language Specification](#).

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Methods](#)

Generics and Arrays (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

In C# 2.0 and later, single-dimensional arrays that have a lower bound of zero automatically implement [IList<T>](#). This enables you to create generic methods that can use the same code to iterate through arrays and other collection types. This technique is primarily useful for reading data in collections. The [IList<T>](#) interface cannot be used to add or remove elements from an array. An exception will be thrown if you try to call an [IList<T>](#) method such as [RemoveAt](#) on an array in this context.

The following code example demonstrates how a single generic method that takes an [IList<T>](#) input parameter can iterate through both a list and an array, in this case an array of integers.

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine(
            "IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Generics](#)
- [Arrays](#)
- [Generics](#)

Generic Delegates (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A [delegate](#) can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
Del<int> m2 = Notify;
```

Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

Code that references the delegate must specify the type argument of the containing class, as follows:

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Generic delegates are especially useful in defining events based on the typical design pattern because the sender argument can be strongly typed and no longer has to be cast to and from [Object](#).

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Generic Methods](#)
- [Generic Classes](#)
- [Generic Interfaces](#)
- [Delegates](#)
- [Generics](#)

Differences Between C++ Templates and C# Generics (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

C# Generics and C++ templates are both language features that provide support for parameterized types. However, there are many differences between the two. At the syntax level, C# generics are a simpler approach to parameterized types without the complexity of C++ templates. In addition, C# does not attempt to provide all of the functionality that C++ templates provide. At the implementation level, the primary difference is that C# generic type substitutions are performed at runtime and generic type information is thereby preserved for instantiated objects. For more information, see [Generics in the Run Time](#).

The following are the key differences between C# Generics and C++ templates:

- C# generics do not provide the same amount of flexibility as C++ templates. For example, it is not possible to call arithmetic operators in a C# generic class, although it is possible to call user defined operators.
- C# does not allow non-type template parameters, such as `template <int i> {}`.
- C# does not support explicit specialization; that is, a custom implementation of a template for a specific type.
- C# does not support partial specialization: a custom implementation for a subset of the type arguments.
- C# does not allow the type parameter to be used as the base class for the generic type.
- C# does not allow type parameters to have default types.
- In C#, a generic type parameter cannot itself be a generic, although constructed types can be used as generics. C++ does allow template parameters.
- C++ allows code that might not be valid for all type parameters in the template, which is then checked for the specific type used as the type parameter. C# requires code in a class to be written in such a way that it will work with any type that satisfies the constraints. For example, in C++ it is possible to write a function that uses the arithmetic operators `+` and `-` on objects of the type parameter, which will produce an error at the time of instantiation of the template with a type that does not support these operators. C# disallows this; the only language constructs allowed are those that can be deduced from the constraints.

See also

- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Templates](#)

Generics in the Run Time (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

When a generic type or method is compiled into Microsoft intermediate language (MSIL), it contains metadata that identifies it as having type parameters. How the MSIL for a generic type is used differs based on whether the supplied type parameter is a value type or reference type.

When a generic type is first constructed with a value type as a parameter, the runtime creates a specialized generic type with the supplied parameter or parameters substituted in the appropriate locations in the MSIL. Specialized generic types are created one time for each unique value type that is used as a parameter.

For example, suppose your program code declared a stack that is constructed of integers:

```
Stack<int> stack;
```

At this point, the runtime generates a specialized version of the `Stack<T>` class that has the integer substituted appropriately for its parameter. Now, whenever your program code uses a stack of integers, the runtime reuses the generated specialized `Stack<T>` class. In the following example, two instances of a stack of integers are created, and they share a single instance of the `Stack<int>` code:

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

However, suppose that another `Stack<T>` class with a different value type such as a `long` or a user-defined structure as its parameter is created at another point in your code. As a result, the runtime generates another version of the generic type and substitutes a `long` in the appropriate locations in MSIL. Conversions are no longer necessary because each specialized generic class natively contains the value type.

Generics work somewhat differently for reference types. The first time a generic type is constructed with any reference type, the runtime creates a specialized generic type with object references substituted for the parameters in the MSIL. Then, every time that a constructed type is instantiated with a reference type as its parameter, regardless of what type it is, the runtime reuses the previously created specialized version of the generic type. This is possible because all references are the same size.

For example, suppose you had two reference types, a `Customer` class and an `Order` class, and also suppose that you created a stack of `Customer` types:

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

At this point, the runtime generates a specialized version of the `Stack<T>` class that stores object references that will be filled in later instead of storing data. Suppose the next line of code creates a stack of another reference type, which is named `Order`:

```
Stack<Order> orders = new Stack<Order>();
```

Unlike with value types, another specialized version of the `Stack<T>` class is not created for the `order` type. Instead, an instance of the specialized version of the `Stack<T>` class is created and the `orders` variable is set to reference it. Suppose that you then encountered a line of code to create a stack of a `Customer` type:

```
customers = new Stack<Customer>();
```

As with the previous use of the `Stack<T>` class created by using the `Order` type, another instance of the specialized `Stack<T>` class is created. The pointers that are contained therein are set to reference an area of memory the size of a `Customer` type. Because the number of reference types can vary wildly from program to program, the C# implementation of generics greatly reduces the amount of code by reducing to one the number of specialized classes created by the compiler for generic classes of reference types.

Moreover, when a generic C# class is instantiated by using a value type or reference type parameter, reflection can query it at runtime and both its actual type and its type parameter can be ascertained.

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Generics](#)

Generics and Reflection (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Because the Common Language Runtime (CLR) has access to generic type information at run time, you can use reflection to obtain information about generic types in the same way as for non-generic types. For more information, see [Generics in the Run Time](#).

In .NET Framework 2.0, several new members were added to the [Type](#) class to enable run-time information for generic types. See the documentation on these classes for more information on how to use these methods and properties. The [System.Reflection.Emit](#) namespace also contains new members that support generics. See [How to: Define a Generic Type with Reflection Emit](#).

For a list of the invariant conditions for terms used in generic reflection, see the [IsGenericType](#) property remarks.

SYSTEM.TYPENAME MEMBER NAME	DESCRIPTION
IsGenericType	Returns true if a type is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments supplied for a constructed type, or the type parameters of a generic type definition.
GetGenericTypeDefinition	Returns the underlying generic type definition for the current constructed type.
GetGenericParameterConstraints	Returns an array of Type objects that represent the constraints on the current generic type parameter.
ContainsGenericParameters	Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types have not been supplied.
GenericParameterAttributes	Gets a combination of GenericParameterAttributes flags that describe the special constraints of the current generic type parameter.
GenericParameterPosition	For a Type object that represents a type parameter, gets the position of the type parameter in the type parameter list of the generic type definition or generic method definition that declared the type parameter.
IsGenericParameter	Gets a value that indicates whether the current Type represents a type parameter of a generic type or method definition.
IsGenericTypeDefinition	Gets a value that indicates whether the current Type represents a generic type definition, from which other generic types can be constructed. Returns true if the type represents the definition of a generic type.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
DeclaringMethod	Returns the generic method that defined the current generic type parameter, or null if the type parameter was not defined by a generic method.
MakeGenericType	Substitutes the elements of an array of types for the type parameters of the current generic type definition, and returns a Type object representing the resulting constructed type.

In addition, members of the [MethodInfo](#) class enable run-time information for generic methods. See the [IsGenericMethod](#) property remarks for a list of invariant conditions for terms used to reflect on generic methods.

SYSTEM.REFLECTION.MEMBERINFO MEMBER NAME	DESCRIPTION
IsGenericMethod	Returns true if a method is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments of a constructed generic method or the type parameters of a generic method definition.
GetGenericMethodDefinition	Returns the underlying generic method definition for the current constructed method.
ContainsGenericParameters	Returns true if the method or any of its enclosing types contain any type parameters for which specific types have not been supplied.
IsGenericMethodDefinition	Returns true if the current MethodInfo represents the definition of a generic method.
MakeGenericMethod	Substitutes the elements of an array of types for the type parameters of the current generic method definition, and returns a MethodInfo object representing the resulting constructed method.

See also

- [C# Programming Guide](#)
- [Generics](#)
- [Reflection and Generic Types](#)
- [Generics](#)

Generics and Attributes (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Attributes can be applied to generic types in the same way as non-generic types. For more information on applying attributes, see [Attributes](#).

Custom attributes are only permitted to reference open generic types, which are generic types for which no type arguments are supplied, and closed constructed generic types, which supply arguments for all type parameters.

The following examples use this custom attribute:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

An attribute can reference an open generic type:

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Specify multiple type parameters using the appropriate number of commas. In this example, `GenericClass2` has two type parameters:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

An attribute can reference a closed constructed generic type:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

An attribute that references a generic type parameter will cause a compile-time error:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
class ClassD<T> { }
```

A generic type cannot inherit from [Attribute](#):

```
//public class CustomAtt<T> : System.Attribute {} //Error
```

To obtain information about a generic type or type parameter at run time, you can use the methods of [System.Reflection](#). For more information, see [Generics and Reflection](#)

See also

- [C# Programming Guide](#)
- [Generics](#)
- [Attributes](#)

Namespaces (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Namespaces are heavily used in C# programming in two ways. First, .NET uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name is not required, as in the following example:

```
using System;
```

```
Console.WriteLine("Hello World!");
```

For more information, see the [using Directive](#).

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the `namespace` keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# [identifier name](#).

Namespaces overview

Namespaces have the following properties:

- They organize large code projects.
- They are delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET `System` namespace.

C# language specification

For more information, see the [Namespaces](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Using Namespaces](#)
- [How to use the My namespace](#)
- [Identifier names](#)
- [using Directive](#)
- [:: Operator](#)

Using namespaces (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

Namespaces are heavily used within C# programs in two ways. Firstly, the .NET classes use namespaces to organize its many classes. Secondly, declaring your own namespaces can help control the scope of class and method names in larger programming projects.

Accessing namespaces

Most C# applications begin with a section of `using` directives. This section lists the namespaces that the application will be using frequently, and saves the programmer from specifying a fully qualified name every time that a method that is contained within is used.

For example, by including the line:

```
using System;
```

At the start of a program, the programmer can use the code:

```
Console.WriteLine("Hello, World!");
```

Instead of:

```
System.Console.WriteLine("Hello, World!");
```

Namespace aliases

You can also use the `using` directive to create an alias for a namespace. Use the `namespace alias qualifier ::` to access the members of the aliased namespace. The following example shows how to create and use a namespace alias:

```
using generics = System.Collections.Generic;

namespace AliasExample
{
    class TestClass
    {
        static void Main()
        {
            generics::Dictionary<string, int> dict = new generics::Dictionary<string, int>()
            {
                ["A"] = 1,
                ["B"] = 2,
                ["C"] = 3
            };

            foreach (string name in dict.Keys)
            {
                System.Console.WriteLine($"{name} {dict[name]}");
            }
            // Output:
            // A 1
            // B 2
            // C 3
        }
    }
}
```

Using namespaces to control scope

The `namespace` keyword is used to declare a scope. The ability to create scopes within your project helps organize code and lets you create globally-unique types. In the following example, a class titled `SampleClass` is defined in two namespaces, one nested inside the other. The `. token` is used to differentiate which method gets called.

```

namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                    "SampleMethod inside NestedNamespace");
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Displays "SampleMethod inside SampleNamespace."
        SampleClass outer = new SampleClass();
        outer.SampleMethod();

        // Displays "SampleMethod inside SampleNamespace."
        SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
        outer2.SampleMethod();

        // Displays "SampleMethod inside NestedNamespace."
        NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
        inner.SampleMethod();
    }
}
}

```

Fully qualified names

Namespaces and types have unique titles described by fully qualified names that indicate a logical hierarchy. For example, the statement `A.B` implies that `A` is the name of the namespace or type, and `B` is nested inside it.

In the following example, there are nested classes and namespaces. The fully qualified name is indicated as a comment following each entity.

```

namespace N1      // N1
{
    class C1      // N1.C1
    {
        class C2  // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2  // N1.N2.C2
        {
        }
    }
}

```

In the previous code segment:

- The namespace `N1` is a member of the global namespace. Its fully qualified name is `N1`.
- The namespace `N2` is a member of `N1`. Its fully qualified name is `N1.N2`.
- The class `C1` is a member of `N1`. Its fully qualified name is `N1.C1`.
- The class name `C2` is used two times in this code. However, the fully qualified names are unique. The first instance of `C2` is declared inside `C1`; therefore, its fully qualified name is: `N1.C1.C2`. The second instance of `C2` is declared inside a namespace `N2`; therefore, its fully qualified name is `N1.N2.C2`.

Using the previous code segment, you can add a new class member, `C3`, to the namespace `N1.N2` as follows:

```

namespace N1.N2
{
    class C3  // N1.N2.C3
    {
    }
}

```

In general, use the [namespace alias qualifier](#) `::` to reference a namespace alias or `global::` to reference the global namespace and `.` to qualify types or members.

It is an error to use `::` with an alias that references a type instead of a namespace. For example:

```
using Alias = System.Console;
```

```

class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}

```

Remember that the word `global` is not a predefined alias; therefore, `global.x` does not have any special meaning. It acquires a special meaning only when it is used with `::`.

Compiler warning CS0440 is generated if you define an alias named global because `global::` always references the global namespace and not an alias. For example, the following line generates the warning:

```
using global = System.Collections; // Warning
```

Using `::` with aliases is a good idea and protects against the unexpected introduction of additional types. For example, consider this example:

```
using Alias = System;
```

```
namespace Library
{
    public class C : Alias::Exception { }
}
```

This works, but if a type named `Alias` were to subsequently be introduced, `Alias.` would bind to that type instead. Using `Alias::Exception` ensures that `Alias` is treated as a namespace alias and not mistaken for a type.

See also

- [C# Programming Guide](#)
- [Namespaces](#)
- [Member access expression](#)
- [:: operator](#)
- [extern alias](#)

How to use the My namespace (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `Microsoft.VisualBasic.MyServices` namespace (`My` in Visual Basic) provides easy and intuitive access to a number of .NET classes, enabling you to write code that interacts with the computer, application, settings, resources, and so on. Although originally designed for use with Visual Basic, the `MyServices` namespace can be used in C# applications.

For more information about using the `MyServices` namespace from Visual Basic, see [Development with My](#).

Add a reference

Before you can use the `MyServices` classes in your solution, you must add a reference to the Visual Basic library.

Add a reference to the Visual Basic library

1. In **Solution Explorer**, right-click the **References** node, and select **Add Reference**.
2. When the **References** dialog box appears, scroll down the list, and select `Microsoft.VisualBasic.dll`.

You might also want to include the following line in the `using` section at the start of your program.

```
using Microsoft.VisualBasic.Devices;
```

Example

This example calls various static methods contained in the `MyServices` namespace. For this code to compile, a reference to `Microsoft.VisualBasic.dll` must be added to the project.

```

using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to network.");
        }
    }
}

```

Not all the classes in the `MyServices` namespace can be called from a C# application: for example, the `FileSystemProxy` class is not compatible. In this particular case, the static methods that are part of `FileSystem`, which are also contained in `VisualBasic.dll`, can be used instead. For example, here is how to use one such method to duplicate a directory:

```

// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");

```

See also

- [C# Programming Guide](#)
- [Namespaces](#)
- [Using Namespaces](#)

XML documentation comments (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

In C#, you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example.

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass {}
```

When you compile with the [DocumentationFile](#) option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as [DocFX](#) or [Sandcastle](#).

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List<T>"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

NOTE

The XML documentation comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

In this section

- [Recommended tags for documentation comments](#)
- [Processing the XML file](#)
- [Delimiters for documentation tags](#)
- [How to use the XML documentation features](#)

Related sections

For more information, see:

- [DocumentationFile \(Process Documentation Comments\)](#)

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)

Recommended tags for documentation comments (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

The C# compiler processes documentation comments in your code and formats them as XML in a file whose name you specify in the `/doc` command-line option. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as [DocFX](#) or [Sandcastle](#).

Tags are processed on code constructs such as types and type members.

NOTE

Documentation comments cannot be applied to a namespace.

The compiler will process any tag that is valid XML. The following tags provide generally used functionality in user documentation.

Tags

<c>	<para>	<see>*	<value>
<code>	<param>*	<seealso>*	
<example>	<paramref>	<summary>	
<exception>*	<permission>*	<typeparam>*	
<include>*	<remarks>	<typeparamref>	
<list>	<inheritdoc>	<returns>	

(* denotes that the compiler verifies syntax.)

If you want angle brackets to appear in the text of a documentation comment, use the HTML encoding of `<` and `>` which is `<` and `>` respectively. This encoding is shown in the following example.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

See also

- [C# programming guide](#)
- [DocumentationFile \(C# compiler options\)](#)
- [XML documentation comments](#)

Process the XML file (C# programming guide)

3/15/2021 • 5 minutes to read • [Edit Online](#)

The compiler generates an ID string for each construct in your code that's tagged to generate documentation. (For information about how to tag your code, see [Recommended Tags for Documentation Comments](#).) The ID string uniquely identifies the construct. Programs that process the XML file can use the ID string to identify the corresponding .NET metadata or reflection item that the documentation applies to.

ID strings

The XML file is not a hierarchical representation of your code. It's a flat list that has a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

- No white space is in the string.
- The first part of the string identifies the kind of member using a single character followed by a colon. The following member types are used:

CHARACTER	MEMBER TYPE	NOTES
N	namespace	You cannot add documentation comments to a namespace, but you can make cref references to them, where supported.
T	type	A type can be a class, interface, struct, enum, or delegate.
F	field	
P	property	Includes indexers or other indexed properties.
M	method	Includes special methods, such as constructors and operators.
E	event	
!	error string	The rest of the string provides information about the error. The C# compiler generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the hash-sign ('#'). It's assumed that no item has a hash-sign directly in its name. For example, the fully qualified name of the String constructor is "System.String.#ctor".
- For properties and methods, the parameter list enclosed in parentheses follows. If there are no

parameters, no parentheses are present. The parameters are separated by commas. The encoding of each parameter follows directly how it's encoded in a .NET signature:

- Base types. Regular types (ELEMENT_TYPE_CLASS or ELEMENT_TYPE_VALUETYPE) are represented as the fully qualified name of the type.
- Intrinsic types (for example, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF, and ELEMENT_TYPE_VOID) are represented as the fully qualified name of the corresponding full type. For example, System.Int32 or System.TypedReference.
- ELEMENT_TYPE_PTR is represented as a '*' following the modified type.
- ELEMENT_TYPE_BYREF is represented as a '@' following the modified type.
- ELEMENT_TYPE_PINNED is represented as a '^' following the modified type. The C# compiler never generates this.
- ELEMENT_TYPE_CMOD_REQ is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this.
- ELEMENT_TYPE_CMOD_OPT is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
- ELEMENT_TYPE_SZARRAY is represented as "[]" following the element type of the array.
- ELEMENT_TYPE_GENERICARRAY is represented as "[?]" following the element type of the array. The C# compiler never generates this.
- ELEMENT_TYPE_ARRAY is represented as [/*lowerbound*.*size*,/*lowerbound*.*size*] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it's omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a 2-dimensional array with 1 as the lower bounds and unspecified sizes is [1;1:].
- ELEMENT_TYPE_FNPTR is represented as "=FUNC:*type* (*signature*)", where *type* is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this.

The following signature components aren't represented because they aren't used to differentiate overloaded methods:

- calling convention
 - return type
 - ELEMENT_TYPE_SENTINEL
- For conversion operators only (`op_Implicit` and `op_Explicit`), the return value of the method is encoded as a '~' followed by the return type.
 - For generic types, the name of the type is followed by a backtick and then a number that indicates the number of generic type parameters. For example:

`<member name="T:SampleClass`2">` is the tag for a type that is defined as `public class SampleClass<T, U>`.

For methods that take generic types as parameters, the generic type parameters are specified as numbers prefaced with backticks (for example `0,`1). Each number represents a zero-based array notation for the type's generic parameters.

Examples

The following examples show how the ID strings for a class and its members are generated:

```
namespace N
{
    ///<summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        ///<summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }

        ///<summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        ///<param name="i">Describe parameter.</param>
        public X(int i) { }

        ///<summary>
        /// Enter description here for field q.
        /// ID string generated is "F:N.X.q".
        /// </summary>
        public string q;

        ///<summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:N.X.PI".
        /// </summary>
        public const double PI = 3.14;

        ///<summary>
        /// Enter description for method f.
        /// ID string generated is "M:N.X.f".
        /// </summary>
        ///<returns>Describe return value.</returns>
        public int f() { return 1; }

        ///<summary>
        /// Enter description for method bb.
        /// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
        /// </summary>
        ///<param name="s">Describe parameter.</param>
        ///<param name="y">Describe parameter.</param>
        ///<param name="z">Describe parameter.</param>
        ///<returns>Describe return value.</returns>
        public int bb(string s, ref int y, void* z) { return 1; }

        ///<summary>
        /// Enter description for method gg.
        /// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        ///<param name="array1">Describe parameter.</param>
        ///<param name="array">Describe parameter.</param>
        ///<returns>Describe return value.</returns>
        public int gg(short[] array1, int[,] array) { return 0; }

        ///<summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op>Addition(N.X,N.X)".
        /// </summary>
        ///<param name="x">Describe parameter.</param>
        ///<param name="xx">Describe parameter.</param>
        ///<returns>Describe return value.</returns>
```

```

...
public static X operator +(X x, X xx) { return x; }

/// <summary>
/// Enter description for property.
/// ID string generated is "P:N.X.prop".
/// </summary>
public int prop { get { return 1; } set { } }

/// <summary>
/// Enter description for event.
/// ID string generated is "E:N.X.d".
/// </summary>
public event D d;

/// <summary>
/// Enter description for property.
/// ID string generated is "P:N.X.Item(System.String)".
/// </summary>
/// <param name="s">Describe parameter.</param>
/// <returns></returns>
public int this[string s] { get { return 1; } }

/// <summary>
/// Enter description for class Nested.
/// ID string generated is "T:N.X.Nested".
/// </summary>
public class Nested { }

/// <summary>
/// Enter description for delegate.
/// ID string generated is "T:N.X.D".
/// </summary>
/// <param name="i">Describe parameter.</param>
public delegate void D(int i);

/// <summary>
/// Enter description for operator.
/// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
/// </summary>
/// <param name="x">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public static explicit operator int(X x) { return 1; }
}

}

```

See also

- [C# programming guide](#)
- [DocumentationFile \(C# compiler options\)](#)
- [XML documentation comments](#)

Delimiters for documentation tags (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

The use of XML doc comments requires delimiters, which indicate to the compiler where a documentation comment begins and ends. You can use the following kinds of delimiters with the XML documentation tags:

- `///`

Single-line delimiter. This is the form that is shown in documentation examples and used by the C# project templates. If there is a white space character following the delimiter, that character is not included in the XML output.

NOTE

The Visual Studio integrated development environment (IDE) automatically inserts the `<summary>` and `</summary>` tags and moves your cursor within these tags after you type the `///` delimiter in the code editor. You can turn this feature on or off in the [Options dialog box](#).

- `/** */`

Multiline delimiters.

There are some formatting rules to follow when you use the `/** */` delimiters:

- On the line that contains the `/**` delimiter, if the remainder of the line is white space, the line is not processed for comments. If the first character after the `/**` delimiter is white space, that white space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
- On the line that contains the `*/` delimiter, if there is only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment.
- For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line. The pattern can consist of optional white space and an asterisk (`*`), followed by more optional white space. If the compiler finds a common pattern at the beginning of each line that does not begin with the `/**` delimiter or the `*/` delimiter, it ignores that pattern for each line.

The following examples illustrate these rules.

- The only part of the following comment that's processed is the line that begins with `<summary>`. The three tag formats produce the same comments.

```
/** <summary>text</summary> */  
  
/**  
 * <summary>text</summary>  
 */  
  
/**  
 * <summary>text</summary>  
 */
```

- The compiler identifies a common pattern of " * " at the beginning of the second and third lines.
The pattern is not included in the output.

```
/**  
 * <summary>  
 * text </summary>*/
```

- The compiler finds no common pattern in the following comment because the second character on the third line is not an asterisk. Therefore, all text on the second and third lines is processed as part of the comment.

```
/**  
 * <summary>  
 * text </summary>  
 */
```

- The compiler finds no pattern in the following comment for two reasons. First, the number of spaces before the asterisk is not consistent. Second, the fifth line begins with a tab, which does not match spaces. Therefore, all text from lines two through five is processed as part of the comment.

```
/**  
 * <summary>  
 * text  
 * text2  
 *   </summary>  
 */
```

See also

- [C# programming guide](#)
- [XML documentation comments](#)
- [DocumentationFile \(C# compiler options\)](#)

How to use the XML documentation features

3/15/2021 • 4 minutes to read • [Edit Online](#)

The following sample provides a basic overview of a type that has been documented.

Example

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
    }
}
```

```

/// <summary>
/// Some other method.
/// </summary>
/// <returns>
/// Return values are described through the returns tag.
/// </returns>
/// <seealso cref="SomeMethod(string)">
/// Notice the use of the cref attribute to reference a specific method.
/// </seealso>
public int SomeOtherMethod()
{
    return 0;
}

public int InterfaceMethod(int n)
{
    return n * n;
}

/// <summary>
/// The entry point for the application.
/// </summary>
/// <param name="args"> A list of command line arguments.</param>
static int Main(System.String[] args)
{
    // TODO: Add code to start application here.
    return 0;
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}

```

The example generates an .xml file with the following contents.

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlesample</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
                Class level summary documentation goes here.
            </summary>
            <remarks>
                Longer comments can be associated with a type or member through
                the remarks tag.
            </remarks>
        </member>
    
```

```
<member name="F:TestClass._name">
    <summary>
        Store for the Name property.
    </summary>
</member>
<member name="M:TestClass.#ctor">
    <summary>
        The class constructor.
    </summary>
</member>
<member name="P:TestClass.Name">
    <summary>
        Name property.
    </summary>
    <value>
        A value tag is used to describe the property value.
    </value>
</member>
<member name="M:TestClass.SomeMethod(System.String)">
    <summary>
        Description for SomeMethod.
    </summary>
    <param name="s"> Parameter description for s goes here.</param>
    <seealso cref="T:System.String">
        You can use the cref attribute on any tag to reference a type or member
        and the compiler will check that the reference exists.
    </seealso>
</member>
<member name="M:TestClass.SomeOtherMethod">
    <summary>
        Some other method.
    </summary>
    <returns>
        Return values are described through the returns tag.
    </returns>
    <seealso cref="M:TestClass.SomeMethod(System.String)">
        Notice the use of the cref attribute to reference a specific method.
    </seealso>
</member>
<member name="M:TestClass.Main(System.String[])>
    <summary>
        The entry point for the application.
    </summary>
    <param name="args"> A list of command line arguments.</param>
</member>
<member name="T:TestInterface">
    <summary>
        Documentation that describes the interface goes here.
    </summary>
    <remarks>
        Details about the interface go here.
    </remarks>
</member>
<member name="M:TestInterface.InterfaceMethod(System.Int32)">
    <summary>
        Documentation that describes the method goes here.
    </summary>
    <param name="n">
        Parameter n requires an integer argument.
    </param>
    <returns>
        The method returns an integer.
    </returns>
</member>
</members>
</doc>
```

Compiling the code

To compile the example, enter the following command:

```
csc XMLsample.cs /doc:XMLsample.xml
```

This command creates the XML file *XMLsample.xml*, which you can view in your browser or by using the [TYPE](#) command.

Robust programming

XML documentation starts with `///`. When you create a new project, the wizards put some starter `///` lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment that says that an error was encountered.
- Developers are free to create their own set of tags. There is a [recommended set of tags](#). Some of the recommended tags have special meanings:
 - The `<param>` tag is used to describe parameters. If used, the compiler verifies that the parameter exists and that all parameters are described in the documentation. If the verification fails, the compiler issues a warning.
 - The `cref` attribute can be attached to any tag to reference a code element. The compiler verifies that this code element exists. If the verification fails, the compiler issues a warning. The compiler respects any `using` statements when it looks for a type described in the `cref` attribute.
 - The `<summary>` tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

NOTE

The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, use the documentation file together with reflection on the actual type or member.

See also

- [C# programming guide](#)
- [DocumentationFile \(C# compiler options\)](#)
- [XML documentation comments](#)
- [DocFX documentation processor](#)
- [Sandcastle documentation processor](#)

<c> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<c>text</c>
```

Parameters

- `text`

The text you would like to indicate as code.

Remarks

The `<c>` tag gives you a way to indicate that text within a description should be marked as code. Use `<code>` to indicate multiple lines as code.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<code> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<code>content</code>
```

Parameters

- `content`

The text you want marked as code.

Remarks

The `<code>` tag is used to indicate multiple lines of code. Use `<c>` to indicate that single-line text within a description should be marked as code.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

See the [<example>](#) article for an example of how to use the `<code>` tag.

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

cref attribute (C# programming guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `cref` attribute in an XML documentation tag means "code reference." It specifies that the inner text of the tag is a code element, such as a type, method, or property. Documentation tools like [DocFX](#) and [Sandcastle](#) use the `cref` attributes to automatically generate hyperlinks to the page where the type or member is documented.

Example

The following example shows `cref` attributes used in `<see>` tags.

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref
        /// attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// </example>
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>
    }
}
```

```
public static T GetGenericValue<T>(T para)
{
    return para;
}

/// <summary>
/// GenericClass.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}" /> type as a cref attribute.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

class Program
{
    static int Main()
    {
        return TestClass.GetZero();
    }
}
```

When compiled, the program produces the following XML file. Notice that the `cref` attribute for the `GetZero` method, for example, has been transformed by the compiler to `"M:TestNamespace.TestClass.GetZero"`. The "M:" prefix means "method" and is a convention that is recognized by documentation tools such as DocFX and Sandcastle. For a complete list of prefixes, see [Processing the XML File](#).

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
                TestClass contains several cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
                This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
                This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/> constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
                The GetZero method.
            </summary>
            <example>
                This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
                <code>
                    class TestClass
                    {
                        static int Main()
                        {
                            return GetZero();
                        }
                    }
                </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue`1(`0)">
            <summary>
                The GetGenericValue method.
            </summary>
            <remarks>
                This sample shows how to specify the <see cref="M:TestNamespace.TestClass.GetGenericValue`1(`0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
                GenericClass.
            </summary>
            <remarks>
                This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref attribute.
            </remarks>
        </member>
    </members>
</doc>

```

See also

- [XML documentation comments](#)
- [Recommended tags for documentation comments](#)

<example> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<example>description</example>
```

Parameters

- `description`

A description of the code sample.

Remarks

The `<example>` tag lets you specify an example of how to use a method or other library member. This commonly involves using the `<code>` tag.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// </example>
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
```

```

        ////
        ///
        /// </code>
        /// </example>
public static int GetZero()
{
    return 0;
}

/// <summary>
/// The GetGenericValue method.
/// </summary>
/// <remarks>
/// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
/// </remarks>

public static T GetGenericValue<T>(T para)
{
    return para;
}

/// <summary>
/// GenericClass.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"> type as a cref attribute.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

class Program
{
    static int Main()
    {
        return TestClass.GetZero();
    }
}

```

See also

- [C# Programming Guide](#)
- [Recommended tags for documentation comments](#)

<exception> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<exception cref="member">description</exception>
```

Parameters

- `cref = "member"`

A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within double quotation marks ("").

For more information on how to format `member` to reference a generic type, see [Processing the XML File](#).

- `description`

A description of the exception.

Remarks

The `<exception>` tag lets you specify which exceptions can be thrown. This tag can be applied to definitions for methods, properties, events, and indexers.

Compile with [DocumentationFile](#) to process documentation comments to a file.

For more information about exception handling, see [Exceptions and Exception Handling](#).

Example

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<include> (C# programming guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Syntax

```
<include file='filename' path='tagpath[@name="id"]' />
```

Parameters

- `filename`

The name of the XML file containing the documentation. The file name can be qualified with a path relative to the source code file. Enclose `filename` in single quotation marks (' ').

- `tagpath`

The path of the tags in `filename` that leads to the tag `name`. Enclose the path in single quotation marks (' ').

- `name`

The name specifier in the tag that precedes the comments; `name` will have an `id`.

- `id`

The ID for the tag that precedes the comments. Enclose the ID in double quotation marks (" ").

Remarks

The `<include>` tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file. By putting the documentation in a separate file, you can apply source control to the documentation separately from the source code. One person can have the source code file checked out and someone else can have the documentation file checked out.

The `<include>` tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your `<include>` use.

Example

This is a multifile example. The following is the first file, which uses `<include>`.

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

The second file, *xml_include_tag.doc*, contains the following documentation comments.

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Program output

The following output is generated when you compile the Test and Test2 classes with the following command line: `-doc:DocFileName.xml`. In Visual Studio, you specify the XML doc comments option in the Build pane of the Project Designer. When the C# compiler sees the `<include>` tag, it searches for documentation comments in *xml_include_tag.doc* instead of the current source file. The compiler then generates *DocFileName.xml*, and this is the file that is consumed by documentation tools such as [Sandcastle](#) to produce the final documentation.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
            </summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
            </summary>
        </member>
    </members>
</doc>
```

See also

- [C# Programming Guide](#)
- [Recommended Tags for Documentation Comments](#)

<inheritdoc> (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<inheritdoc/>
```

InheritDoc

Inherit XML comments from base classes, interfaces, and similar methods. This eliminates unwanted copying and pasting of duplicate XML comments and automatically keeps XML comments synchronized.

Remarks

Add your XML comments in base classes or interfaces and let InheritDoc copy the comments to implementing classes.

Add your XML comments to your synchronous methods and let InheritDoc copy the comments to your asynchronous versions of the same methods.

If you want to copy the comments from a specific member you can use the `cref` attribute to specify the member.

Examples

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
public class MainClass
{
}

///<inheritdoc/>
public class TestClass: MainClass
{
```

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this interface.
/// </summary>
public interface ITestInterface
{
}

///<inheritdoc cref="ITestInterface"/>
public class TestClass : ITestInterface
{
```

See also

- [C# Programming Guide](#)
- [Recommended Tags for Documentation Comments](#)

<list> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<list type="bullet|number|table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

Parameters

- `term`

A term to define, which will be defined in `description`.

- `description`

Either an item in a bullet or numbered list or the definition of a `term`.

Remarks

The `<listheader>` block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for `term` in the heading.

Each item in the list is specified with an `<item>` block. When creating a definition list, you will need to specify both `term` and `description`. However, for a table, bulleted list, or numbered list, you only need to supply an entry for `description`.

A list or table can have as many `<item>` blocks as needed.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <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>
    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<para> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<para>content</para>
```

Parameters

- `content`

The text of the paragraph.

Remarks

The `<para>` tag is for use inside a tag, such as `<summary>`, `<remarks>`, or `<returns>`, and lets you add structure to the text.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

See `<summary>` for an example of using `<para>`.

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<param> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<param name="name">description</param>
```

Parameters

- `name`

The name of a method parameter. Enclose the name in double quotation marks (" ").

- `description`

A description for the parameter.

Remarks

The `<param>` tag should be used in the comment for a method declaration to describe one of the parameters for the method. To document multiple parameters, use multiple `<param>` tags.

The text for the `<param>` tag is displayed in IntelliSense, the Object Browser, and the Code Comment Web Report.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<paramref> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<paramref name="name"/>
```

Parameters

- `name`

The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

Remarks

The `<paramref>` tag gives you a way to indicate that a word in the code comments, for example in a `<summary>` or `<remarks>` block refers to a parameter. The XML file can be processed to format this word in some distinct way, such as with a bold or italic font.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# Programming Guide](#)
- [Recommended tags for documentation comments](#)

<permission> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<permission cref="member">description</permission>
```

Parameters

- `cref = "member"`

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates `member` to the canonical element name in the output XML. *member* must appear within double quotation marks (" ").

For information on how to create a cref reference to a generic type, see [cref attribute](#).

- `description`

A description of the access to the member.

Remarks

The `<permission>` tag lets you document the access of a member. The [PermissionSet](#) class lets you specify access to a member.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<remarks> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<remarks>description</remarks>
```

Parameters

- [Description](#)

A description of the member.

Remarks

The `<remarks>` tag is used to add information about a type, supplementing the information specified with `<summary>`. This information is displayed in the Object Browser window.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# Programming Guide](#)
- [Recommended tags for documentation comments](#)

<returns> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<returns>description</returns>
```

Parameters

- `description`

A description of the return value.

Remarks

The `<returns>` tag should be used in the comment for a method declaration to describe the return value.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<see> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<see cref="member"/>
```

Parameters

- `cref = "member"`

A reference to a member or field that is available to be called from the current compilation environment.

The compiler checks that the given code element exists and passes `member` to the element name in the output XML. Place *member* within double quotation marks ("").

Remarks

The `<see>` tag lets you specify a link from within text. Use `<seealso>` to indicate that text should be placed in a See Also section. Use the [cref Attribute](#) to create internal hyperlinks to documentation pages for code elements. Also, `href` is a valid Attribute that will function as a hyperlink.

Compile with [DocumentationFile](#) to process documentation comments to a file.

The following example shows a `<see>` tag within a summary section.

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
    cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<seealso> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<seealso cref="member"/>
```

Parameters

- `cref = "member"`

A reference to a member or field that is available to be called from the current compilation environment.

The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within double quotation marks ("").

For information on how to create a cref reference to a generic type, see [cref attribute](#).

Remarks

The `<seealso>` tag lets you specify the text that you might want to appear in a See Also section. Use `<see>` to specify a link from within text.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

See [`<summary>`](#) for an example of using `<seealso>`.

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<summary> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<summary>description</summary>
```

Parameters

- `description`

A summary of the object.

Remarks

The `<summary>` tag should be used to describe a type or a type member. Use `<remarks>` to add supplemental information to a type description. Use the [eref Attribute](#) to enable documentation tools such as [DocFX](#) and [Sandcastle](#) to create internal hyperlinks to documentation pages for code elements.

The text for the `<summary>` tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser Window.

Compile with [DocumentationFile](#) to process documentation comments to a file. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as [DocFX](#) or [Sandcastle](#).

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
    cref="System.Console.WriteLine(System.String)" /> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

The previous example produces the following XML file.

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>YourNamespace</name>
  </assembly>
  <members>
    <member name="T:TestClass">
      text for class TestClass
    </member>
    <member name="M:TestClass.DoWork(System.Int32)">
      <summary>DoWork is a method in the TestClass class.
      <para>Here's how you could make a second paragraph in a description. <see
      cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
      </summary>
      <seealso cref="M:TestClass.Main"/>
    </member>
    <member name="M:TestClass.Main">
      text for Main
    </member>
  </members>
</doc>

```

cref example

The following example shows how to make a `cref` reference to a generic type.

```

// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }

class Program
{
    static void Main() { }
}

```

The previous example produces the following XML file.

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>CRefTest</name>
  </assembly>
  <members>
    <member name="T:A">
      <summary cref="T:C`1">
        </summary>
    </member>
    <member name="T:B">
      <summary cref="T:C`1">
        </summary>
    </member>
    <member name="T:C`1">
      <summary cref="T:A">
        </summary>
      <typeparam name="T"></typeparam>
    </member>
  </members>
</doc>
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<typeparam> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<typeparam name="name">description</typeparam>
```

Parameters

- `name`

The name of the type parameter. Enclose the name in double quotation marks (" ").

- `description`

A description for the type parameter.

Remarks

The `<typeparam>` tag should be used in the comment for a generic type or method declaration to describe a type parameter. Add a tag for each type parameter of the generic type or method.

For more information, see [Generics](#).

The text for the `<typeparam>` tag will be displayed in IntelliSense, the [Object Browser Window](#) code comment web report.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

See also

- [C# reference](#)
- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<typeparamref> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<typeparamref name="name"/>
```

Parameters

- `name`

The name of the type parameter. Enclose the name in double quotation marks ("").

Remarks

For more information on type parameters in generic types and methods, see [Generics](#).

Use this tag to enable consumers of the documentation file to format the word in some distinct way, for example in italics.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

<value> (C# programming guide)

3/15/2021 • 2 minutes to read • [Edit Online](#)

Syntax

```
<value>property-description</value>
```

Parameters

- `property-description`

A description for the property.

Remarks

The `<value>` tag lets you describe the value that a property represents. When you add a property via code wizard in the Visual Studio .NET development environment, it adds a `<summary>` tag for the new property. You should then manually add a `<value>` tag to describe the value that the property represents.

Compile with [DocumentationFile](#) to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

See also

- [C# programming guide](#)
- [Recommended tags for documentation comments](#)

Exceptions and Exception Handling (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the `try`, `catch`, and `finally` keywords to try actions that may not succeed, to handle failures when you decide that it's reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by .NET or third-party libraries, or by application code. Exceptions are created by using the `throw` keyword.

In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When an exception is thrown happens, the CLR will unwind the stack, looking for a method with a `catch` block for the specific exception type, and it will execute the first such `catch` block that it finds. If it finds no appropriate `catch` block anywhere in the call stack, it will terminate the process and display a message to the user.

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a `DivideByZeroException was unhandled` error.

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from `System.Exception`.
- Use a `try` block around the statements that might throw exceptions.
- Once an exception occurs in the `try` block, the flow of control jumps to the first associated exception

handler that is present anywhere in the call stack. In C#, the `catch` keyword is used to define an exception handler.

- If no exception handler for a given exception is present, the program stops executing with an error message.
- Don't catch an exception unless you can handle it and leave the application in a known state. If you catch `System.Exception`, rethrow it using the `throw` keyword at the end of the `catch` block.
- If a `catch` block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the `throw` keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a `finally` block is executed even if an exception is thrown. Use a `finally` block to release resources, for example to close any streams or files that were opened in the `try` block.
- Managed exceptions in .NET are implemented on top of the Win32 structured exception handling mechanism. For more information, see [Structured Exception Handling \(C/C++\)](#) and [A Crash Course on the Depths of Win32 Structured Exception Handling](#).

C# Language Specification

For more information, see [Exceptions](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [SystemException](#)
- [C# Keywords](#)
- [throw](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [Exceptions](#)

Use exceptions (C# programming guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

In C#, errors in the program at run time are propagated through the program by using a mechanism called exceptions. Exceptions are thrown by code that encounters an error and caught by code that can correct the error. Exceptions can be thrown by the .NET runtime or by code in a program. Once an exception is thrown, it propagates up the call stack until a `catch` statement for the exception is found. Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.

Exceptions are represented by classes derived from [Exception](#). This class identifies the type of exception and contains properties that have details about the exception. Throwing an exception involves creating an instance of an exception-derived class, optionally configuring properties of the exception, and then throwing the object by using the `throw` keyword. For example:

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

After an exception is thrown, the runtime checks the current statement to see whether it is within a `try` block. If it is, any `catch` blocks associated with the `try` block are checked to see whether they can catch the exception. `catch` blocks typically specify exception types; if the type of the `catch` block is the same type as the exception, or a base class of the exception, the `catch` block can handle the method. For example:

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

If the statement that throws an exception isn't within a `try` block or if the `try` block that encloses it has no matching `catch` block, the runtime checks the calling method for a `try` statement and `catch` blocks. The runtime continues up the calling stack, searching for a compatible `catch` block. After the `catch` block is found and executed, control is passed to the next statement after that `catch` block.

A `try` statement can contain more than one `catch` block. The first `catch` statement that can handle the exception is executed; any following `catch` statements, even if they're compatible, are ignored. Catch blocks should always be ordered from most specific (or most-derived) to least specific. For example:

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
    }
}
```

Before the `catch` block is executed, the runtime checks for `finally` blocks. `Finally` blocks enable the programmer to clean up any ambiguous state that could be left over from an aborted `try` block, or to release any external resources (such as graphics handles, database connections, or file streams) without waiting for the garbage collector in the runtime to finalize the objects. For example:

```

static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xFF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}

```

If `WriteByte()` threw an exception, the code in the second `try` block that tries to reopen the file would fail if `file.Close()` isn't called, and the file would remain locked. Because `finally` blocks are executed even if an exception is thrown, the `finally` block in the previous example allows for the file to be closed correctly and helps avoid an error.

If no compatible `catch` block is found on the call stack after an exception is thrown, one of three things occurs:

- If the exception is within a finalizer, the finalizer is aborted and the base finalizer, if any, is called.
- If the call stack contains a static constructor, or a static field initializer, a `TypeInitializationException` is thrown, with the original exception assigned to the `InnerException` property of the new exception.
- If the start of the thread is reached, the thread is terminated.

Exception Handling (C# Programming Guide)

3/6/2021 • 4 minutes to read • [Edit Online](#)

A `try` block is used by C# programmers to partition code that might be affected by an exception. Associated `catch` blocks are used to handle any resulting exceptions. A `finally` block contains code that is run whether or not an exception is thrown in the `try` block, such as releasing resources that are allocated in the `try` block. A `try` block requires one or more associated `catch` blocks, or a `finally` block, or both.

The following examples show a `try-catch` statement, a `try-finally` statement, and a `try-catch-finally` statement.

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

A `try` block without a `catch` or `finally` block causes a compiler error.

Catch Blocks

A `catch` block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from `Exception`. In general, don't specify `Exception` as the exception filter unless either you know how to handle all exceptions that might be thrown in the `try` block, or you've included a `throw` statement at the end of your `catch` block.

Multiple `catch` blocks with different exception classes can be chained together. The `catch` blocks are evaluated from top to bottom in your code, but only one `catch` block is executed for each exception that is thrown. The first `catch` block that specifies the exact type or a base class of the thrown exception is executed. If no `catch` block specifies a matching exception class, a `catch` block that doesn't have any type is selected, if one is present in the statement. It's important to position `catch` blocks with the most specific (that is, the most derived) exception classes first.

Catch exceptions when the following conditions are true:

- You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a [FileNotFoundException](#) object.
- You can create and throw a new, more specific exception.

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "Parameter index is out of range.", e);
    }
}
```

- You want to partially handle an exception before passing it on for additional handling. In the following example, a `catch` block is used to add an entry to an error log before rethrowing the exception.

```
try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

You can also specify *exception filters* to add a boolean expression to a catch clause. These indicate that a specific catch clause matches only when that condition is true. In the following example, both catch clauses use the same exception class, but an additional condition is checked to create a different error message:

```

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "Parameter index cannot be greater than the array size.", e);
    }
}

```

An exception filter that always returns `false` can be used to examine all exceptions but not process them. A typical use is to log exceptions:

```

public static void Main()
{
    try
    {
        string? s = null;
        Console.WriteLine(s.Length);
    }
    catch (Exception e) when (LogException(e))
    {
    }
    Console.WriteLine("Exception must have been handled");
}

private static bool LogException(Exception e)
{
    Console.WriteLine($"\\tIn the log routine. Caught {e.GetType()}");
    Console.WriteLine($"\\tMessage: {e.Message}");
    return false;
}

```

The `LogException` method always returns `false`, no `catch` clause using this exception filter matches. The `catch` clause can be general, using `System.Exception`, and later clauses can process more specific exception classes.

Finally Blocks

A `finally` block enables you to clean up actions that are performed in a `try` block. If present, the `finally` block executes last, after the `try` block and any matched `catch` block. A `finally` block always runs, whether an exception is thrown or a `catch` block matching the exception type is found.

The `finally` block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects. For more information See the [using Statement](#).

In the following example, the `finally` block is used to close a file that is opened in the `try` block. Notice that the state of the file handle is checked before the file is closed. If the `try` block can't open the file, the file handle still has the value `null` and the `finally` block doesn't try to close it. Instead, if the file is opened successfully in the `try` block, the `finally` block closes the open file.

```
FileStream? file = null;
FileInfo fileInfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

C# Language Specification

For more information, see [Exceptions](#) and [The try statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [using Statement](#)

Creating and Throwing Exceptions (C# Programming Guide)

3/6/2021 • 3 minutes to read • [Edit Online](#)

Exceptions are used to indicate that an error has occurred while running the program. Exception objects that describe an error are created and then *thrown* with the `throw` keyword. The runtime then searches for the most compatible exception handler.

Programmers should throw exceptions when one or more of the following conditions are true:

- The method can't complete its defined functionality. For example, if a parameter to a method has an invalid value:

```
static void CopyObject(SampleClass original)
{
    _ = original ?? throw new ArgumentException("Parameter cannot be null", nameof(original));
}
```

- An inappropriate call to an object is made, based on the object state. One example might be trying to write to a read-only file. In cases where an object state doesn't allow an operation, throw an instance of `InvalidOperationException` or an object based on a derivation of this class. The following code is an example of a method that throws an `InvalidOperationException` object:

```
public class ProgramLog
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new InvalidOperationException("Logfile cannot be read-only");
        }
        // Else write data to the log and return.
    }
}
```

- When an argument to a method causes an exception. In this case, the original exception should be caught and an `ArgumentException` instance should be created. The original exception should be passed to the constructor of the `ArgumentException` as the `InnerException` parameter:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index), ex);
    }
}
```

Exceptions contain a property named [StackTrace](#). This string contains the name of the methods on the current call stack, together with the file name and line number where the exception was thrown for each method. A [StackTrace](#) object is created automatically by the common language runtime (CLR) from the point of the `throw` statement, so that exceptions must be thrown from the point where the stack trace should begin.

All exceptions contain a property named [Message](#). This string should be set to explain the reason for the exception. Information that is sensitive to security shouldn't be put in the message text. In addition to [Message](#), [ArgumentException](#) contains a property named [ParamName](#) that should be set to the name of the argument that caused the exception to be thrown. In a property setter, [ParamName](#) should be set to `value`.

Public and protected methods throw exceptions whenever they can't complete their intended functions. The exception class thrown is the most specific exception available that fits the error conditions. These exceptions should be documented as part of the class functionality, and derived classes or updates to the original class should retain the same behavior for backward compatibility.

Things to Avoid When Throwing Exceptions

The following list identifies practices to avoid when throwing exceptions:

- Don't use exceptions to change the flow of a program as part of ordinary execution. Use exceptions to report and handle error conditions.
- Exceptions shouldn't be returned as a return value or parameter instead of being thrown.
- Don't throw [System.Exception](#), [System.SystemException](#), [System.NullReferenceException](#), or [System.IndexOutOfRangeException](#) intentionally from your own source code.
- Don't create exceptions that can be thrown in debug mode but not release mode. To identify run-time errors during the development phase, use Debug Assert instead.

Defining Exception Classes

Programs can throw a predefined exception class in the [System](#) namespace (except where previously noted), or create their own exception classes by deriving from [Exception](#). The derived classes should define at least four constructors: one parameterless constructor, one that sets the [Message](#) property, and one that sets both the [Message](#) and [InnerException](#) properties. The fourth constructor is used to serialize the exception. New exception classes should be serializable. For example:

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

Add new properties to the exception class when the data they provide is useful to resolving the exception. If new properties are added to the derived exception class, `Tostring()` should be overridden to return the added information.

C# Language Specification

For more information, see [Exceptions](#) and [The throw statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Exception Hierarchy](#)

Compiler-Generated Exceptions (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

Some exceptions are thrown automatically by the .NET runtime when basic operations fail. These exceptions and their error conditions are listed in the following table.

EXCEPTION	DESCRIPTION
ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException .
ArrayTypeMismatchException	Thrown when an array can't store a given element because the actual type of the element is incompatible with the actual type of the array.
DivideByZeroException	Thrown when an attempt is made to divide an integral value by zero.
IndexOutOfRangeException	Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
InvalidCastException	Thrown when an explicit conversion from a base type to an interface or to a derived type fails at runtime.
NullReferenceException	Thrown when an attempt is made to reference an object whose value is <code>null</code> .
OutOfMemoryException	Thrown when an attempt to allocate memory using the <code>new</code> operator fails. This exception indicates that the memory available to the common language runtime has been exhausted.
OverflowException	Thrown when an arithmetic operation in a <code>checked</code> context overflows.
StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.
TypeInitializationException	Thrown when a static constructor throws an exception and no compatible <code>catch</code> clause exists to catch it.

See also

- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

How to handle an exception using try/catch (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The purpose of a `try-catch` block is to catch and handle an exception generated by working code. Some exceptions can be handled in a `catch` block and the problem solved without the exception being rethrown; however, more often the only thing that you can do is make sure that the appropriate exception is thrown.

Example

In this example, `IndexOutOfRangeException` isn't the most appropriate exception:

`ArgumentOutOfRangeException` makes more sense for the method because the error is caused by the `index` argument passed in by the caller.

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of range.", e);
    }
}
```

Comments

The code that causes an exception is enclosed in the `try` block. A `catch` statement is added immediately after to handle `IndexOutOfRangeException`, if it occurs. The `catch` block handles the `IndexOutOfRangeException` and throws the more appropriate `ArgumentOutOfRangeException` exception instead. In order to provide the caller with as much information as possible, consider specifying the original exception as the `InnerException` of the new exception. Because the `InnerException` property is `read-only`, you must assign it in the constructor of the new exception.

How to execute cleanup code using finally (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The purpose of a `finally` statement is to ensure that the necessary cleanup of objects, usually objects that are holding external resources, occurs immediately, even if an exception is thrown. One example of such cleanup is calling `Close` on a `FileStream` immediately after use instead of waiting for the object to be garbage collected by the common language runtime, as follows:

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Example

To turn the previous code into a `try-catch-finally` statement, the cleanup code is separated from the working code, as follows.

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        file?.Close();
    }
}
```

Because an exception can occur at any time within the `try` block before the `OpenWrite()` call, or the `OpenWrite()` call itself could fail, we aren't guaranteed that the file is open when we try to close it. The `finally` block adds a check to make sure that the `FileStream` object isn't `null` before you call the `Close` method. Without the `null` check, the `finally` block could throw its own `NullReferenceException`, but throwing exceptions in `finally` blocks should be avoided if it's possible.

A database connection is another good candidate for being closed in a `finally` block. Because the number of connections allowed to a database server is sometimes limited, you should close database connections as quickly as possible. If an exception is thrown before you can close your connection, using the `finally` block is better than waiting for garbage collection.

See also

- [using Statement](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

How to catch a non-CLS exception

11/2/2020 • 2 minutes to read • [Edit Online](#)

Some .NET languages, including C++/CLI, allow objects to throw exceptions that do not derive from [Exception](#). Such exceptions are called *non-CLS exceptions* or *non-Exceptions*. In C# you cannot throw non-CLS exceptions, but you can catch them in two ways:

- Within a `catch (RuntimeWrappedException e)` block.

By default, a Visual C# assembly catches non-CLS exceptions as wrapped exceptions. Use this method if you need access to the original exception, which can be accessed through the [RuntimeWrappedException.WrappedException](#) property. The procedure later in this topic explains how to catch exceptions in this manner.

- Within a general catch block (a catch block without an exception type specified) that is put after all other `catch` blocks.

Use this method when you want to perform some action (such as writing to a log file) in response to non-CLS exceptions, and you do not need access to the exception information. By default the common language runtime wraps all exceptions. To disable this behavior, add this assembly-level attribute to your code, typically in the AssemblyInfo.cs file:

```
[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)].
```

To catch a non-CLS exception

Within a `catch(RuntimeWrappedException e)` block, access the original exception through the [RuntimeWrappedException.WrappedException](#) property.

Example

The following example shows how to catch a non-CLS exception that was thrown from a class library written in C++/CLI. Note that in this example, the C# client code knows in advance that the exception type being thrown is a [System.String](#). You can cast the [RuntimeWrappedException.WrappedException](#) property back its original type as long as that type is accessible from your code.

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

See also

- [RuntimeWrappedException](#)
- [Exceptions and Exception Handling](#)

File system and the registry (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following articles show how to use C# and .NET to perform various basic operations on files, folders, and the registry.

In this section

TITLE	DESCRIPTION
How to iterate through a directory tree	Shows how to manually iterate through a directory tree.
How to get information about files, folders, and drives	Shows how to retrieve information such as creation times and size, about files, folders and drives.
How to create a file or folder	Shows how to create a new file or folder.
How to copy, delete, and move files and folders (C# Programming Guide)	Shows how to copy, delete and move files and folders.
How to provide a progress dialog box for file operations	Shows how to display a standard Windows progress dialog for certain file operations.
How to write to a text file	Shows how to write to a text file.
How to read from a text file	Shows how to read from a text file.
How to read a text file one line at a time	Shows how to retrieve text from a file one line at a time.
How to create a key in the registry	Shows how to write a key to the system registry.

Related sections

- [File and Stream I/O](#)
- [How to copy, delete, and move files and folders \(C# Programming Guide\)](#)
- [C# Programming Guide](#)
- [System.IO](#)

How to iterate through a directory tree (C# Programming Guide)

11/2/2020 • 7 minutes to read • [Edit Online](#)

The phrase "iterate a directory tree" means to access each file in each nested subdirectory under a specified root folder, to any depth. You do not necessarily have to open each file. You can just retrieve the name of the file or subdirectory as a `string`, or you can retrieve additional information in the form of a [System.IO.FileInfo](#) or [System.IO.DirectoryInfo](#) object.

NOTE

In Windows, the terms "directory" and "folder" are used interchangeably. Most documentation and user interface text uses the term "folder," but .NET class libraries use the term "directory."

In the simplest case, in which you know for certain that you have access permissions for all directories under a specified root, you can use the `System.IO.SearchOption.AllDirectories` flag. This flag returns all the nested subdirectories that match the specified pattern. The following example shows how to use this flag.

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

The weakness in this approach is that if any one of the subdirectories under the specified root causes a [DirectoryNotFoundException](#) or [UnauthorizedAccessException](#), the whole method fails and returns no directories. The same is true when you use the [GetFiles](#) method. If you have to handle these exceptions on specific subfolders, you must manually walk the directory tree, as shown in the following examples.

When you manually walk a directory tree, you can handle the subdirectories first (*pre-order traversal*), or the files first (*post-order traversal*). If you perform a pre-order traversal, you walk the whole tree under the current folder before iterating through the files that are directly in that folder itself. The examples later in this document perform post-order traversal, but you can easily modify them to perform pre-order traversal.

Another option is whether to use recursion or a stack-based traversal. The examples later in this document show both approaches.

If you have to perform a variety of operations on files and folders, you can modularize these examples by refactoring the operation into separate functions that you can invoke by using a single delegate.

NOTE

NTFS file systems can contain *reparse points* in the form of *junction points*, *symbolic links*, and *hard links*. .NET methods such as [GetFiles](#) and [GetDirectories](#) will not return any subdirectories under a reparse point. This behavior guards against the risk of entering into an infinite loop when two reparse points refer to each other. In general, you should use extreme caution when you deal with reparse points to ensure that you do not unintentionally modify or delete files. If you require precise control over reparse points, use platform invoke or native code to call the appropriate Win32 file system methods directly.

Example

The following example shows how to walk a directory tree by using recursion. The recursive approach is elegant

but has the potential to cause a stack overflow exception if the directory tree is large and deeply nested.

The particular exceptions that are handled, and the particular actions that are performed on each file or folder, are provided as examples only. You should modify this code to meet your specific requirements. See the comments in the code for more information.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

```
if (files != null)
{
    foreach (System.IO.FileInfo fi in files)
    {
        // In this example, we only access the existing FileInfo object. If we
        // want to open, delete or modify the file, then
        // a try-catch block is required here to handle the case
        // where the file has been deleted since the call to TraverseTree().
        Console.WriteLine(fi.FullName);
    }

    // Now find all the subdirectories under this directory.
    subDirs = root.GetDirectories();

    foreach (System.IO.DirectoryInfo dirInfo in subDirs)
    {
        // Recursive call for each subdirectory.
        WalkDirectoryTree(dirInfo);
    }
}
```

Example

The following example shows how to iterate through files and folders in a directory tree without using recursion. This technique uses the generic `Stack<T>` collection type, which is a last in first out (LIFO) stack.

The particular exceptions that are handled, and the particular actions that are performed on each file or folder, are provided as examples only. You should modify this code to meet your specific requirements. See the comments in the code for more information.

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
        }
    }
}
```

```

// discovery permission on a folder or file. It may or may not be acceptable
// to ignore the exception and continue enumerating the remaining files and
// folders. It is also possible (but unlikely) that a DirectoryNotFoundException
// will be raised. This will happen if currentDir has been deleted by
// another application or thread after our call to Directory.Exists. The
// choice of which exceptions to catch depends entirely on the specific task
// you are intending to perform and also on how much you know with certainty
// about the systems on which this code will run.
catch (UnauthorizedAccessException e)
{
    Console.WriteLine(e.Message);
    continue;
}
catch (System.IO.DirectoryNotFoundException e)
{
    Console.WriteLine(e.Message);
    continue;
}

string[] files = null;
try
{
    files = System.IO.Directory.GetFiles(currentDir);
}

catch (UnauthorizedAccessException e)
{
    Console.WriteLine(e.Message);
    continue;
}

catch (System.IO.DirectoryNotFoundException e)
{
    Console.WriteLine(e.Message);
    continue;
}

// Perform the required action on each file here.
// Modify this block to perform your required task.
foreach (string file in files)
{
    try
    {
        // Perform whatever action is required in your scenario.
        System.IO.FileInfo fi = new System.IO.FileInfo(file);
        Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // If file was deleted by a separate application
        // or thread since the call to TraverseTree()
        // then just continue.
        Console.WriteLine(e.Message);
        continue;
    }
}

// Push the subdirectories onto the stack for traversal.
// This could also be done before handing the files.
foreach (string str in subDirs)
    dirs.Push(str);
}
}
}

```

It is generally too time-consuming to test every folder to determine whether your application has permission to open it. Therefore, the code example just encloses that part of the operation in a `try/catch` block. You can

modify the `catch` block so that when you are denied access to a folder, you try to elevate your permissions and then access it again. As a rule, only catch those exceptions that you can handle without leaving your application in an unknown state.

If you must store the contents of a directory tree, either in memory or on disk, the best option is to store only the [FullName](#) property (of type `string`) for each file. You can then use this string to create a new [FileInfo](#) or [DirectoryInfo](#) object as necessary, or open any file that requires additional processing.

Robust Programming

Robust file iteration code must take into account many complexities of the file system. For more information on the Windows file system, see [NTFS overview](#).

See also

- [System.IO](#)
- [LINQ and File Directories](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to get information about files, folders, and drives (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

In .NET, you can access file system information by using the following classes:

- [System.IO.FileInfo](#)
- [System.IO.DirectoryInfo](#)
- [System.IO.DriveInfo](#)
- [System.IO.Directory](#)
- [System.IO.File](#)

The [FileInfo](#) and [DirectoryInfo](#) classes represent a file or directory and contain properties that expose many of the file attributes that are supported by the NTFS file system. They also contain methods for opening, closing, moving, and deleting files and folders. You can create instances of these classes by passing a string that represents the name of the file, folder, or drive in to the constructor:

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

You can also obtain the names of files, folders, or drives by using calls to [DirectoryInfo.GetDirectories](#), [DirectoryInfo.GetFiles](#), and [DriveInfo.RootDirectory](#).

The [System.IO.Directory](#) and [System.IO.File](#) classes provide static methods for retrieving information about directories and files.

Example

The following example shows various ways to access information about files and folders.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }

        // Get the subdirectories directly that is under the root.
    }
}
```

```

// See "How to: Iterate Through a Directory Tree" for an example of how to
// iterate through an entire tree.
System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories(".*");

foreach (System.IO.DirectoryInfo d in dirInfos)
{
    Console.WriteLine(d.Name);
}

// The Directory and File classes provide several static methods
// for accessing files and directories.

// Get the current application directory.
string currentDirName = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(currentDirName);

// Get an array of file names as strings rather than FileInfo objects.
// Use this method when storage space is an issue, and when you might
// hold on to the file name reference for a while before you try to access
// the file.
string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

foreach (string s in files)
{
    // Create the FileInfo object only when needed to ensure
    // the information is as current as possible.
    System.IO.FileInfo fi = null;
    try
    {
        fi = new System.IO.FileInfo(s);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // To inform the user and continue is
        // sufficient for this demonstration.
        // Your application may require different behavior.
        Console.WriteLine(e.Message);
        continue;
    }
    Console.WriteLine("{0} : {1}", fi.Name, fi.Directory);
}

// Change the directory. In this case, first check to see
// whether it already exists, and create it if it does not.
// If this is not appropriate for your application, you can
// handle the System.IO.IOException that will be raised if the
// directory cannot be found.
if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\")) {
    System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");

    System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

    currentDirName = System.IO.Directory.GetCurrentDirectory();
    Console.WriteLine(currentDirName);

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

```

Robust Programming

When you process user-specified path strings, you should also handle exceptions for the following conditions:

- The file name is malformed. For example, it contains invalid characters or only white space.
- The file name is null.
- The file name is longer than the system-defined maximum length.
- The file name contains a colon (:).

If the application does not have sufficient permissions to read the specified file, the `Exists` method returns `false` regardless of whether a path exists; the method does not throw an exception.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to create a file or folder (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

You can programmatically create a folder on your computer, create a subfolder, create a file in the subfolder, and write data to the file.

Example

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
        }
    }
}
```

```

        }

    }
    else
    {
        Console.WriteLine("File \'{0}\' already exists.", fileName);
        return;
    }

    // Read and display the data from your file.
    try
    {
        byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
        foreach (byte b in readBuffer)
        {
            Console.Write(b + " ");
        }
        Console.WriteLine();
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
// Sample output:

// Path to my file: c:\Top-Level Folder\SubFolder\ttxvause.vv0

//0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
//30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
// 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
//3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}

```

If the folder already exists, [CreateDirectory](#) does nothing, and no exception is thrown. However, [File.Create](#) replaces an existing file with a new file. The example uses an `if - else` statement to prevent an existing file from being replaced.

By making the following changes in the example, you can specify different outcomes based on whether a file with a certain name already exists. If such a file doesn't exist, the code creates one. If such a file exists, the code appends data to that file.

- Specify a non-random file name.

```

// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";

```

- Replace the `if - else` statement with the `using` statement in the following code.

```

using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}

```

Run the example several times to verify that data is added to the file each time.

For more `FileMode` values that you can try, see [FileMode](#).

The following conditions may cause an exception:

- The folder name is malformed. For example, it contains illegal characters or is only white space ([ArgumentException](#) class). Use the [Path](#) class to create valid path names.
- The parent folder of the folder to be created is read-only ([IOException](#) class).
- The folder name is `null` ([ArgumentNullException](#) class).
- The folder name is too long ([PathTooLongException](#) class).
- The folder name is only a colon, ":" ([PathTooLongException](#) class).

.NET Security

An instance of the [SecurityException](#) class may be thrown in partial-trust situations.

If you don't have permission to create the folder, the example throws an instance of the [UnauthorizedAccessException](#) class.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to copy, delete, and move files and folders (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The following examples show how to copy, move, and delete files and folders in a synchronous manner by using the [System.IO.File](#), [System.IO.Directory](#), [System.IO.FileInfo](#), and [System.IO.DirectoryInfo](#) classes from the [System.IO](#) namespace. These examples do not provide a progress bar or any other user interface. If you want to provide a standard progress dialog box, see [How to provide a progress dialog box for file operations](#).

Use [System.IO.FileSystemWatcher](#) to provide events that will enable you to calculate the progress when operating on multiple files. Another approach is to use platform invoke to call the relevant file-related methods in the Windows Shell. For information about how to perform these file operations asynchronously, see [Asynchronous File I/O](#).

Example

The following example shows how to copy files and directories.

```

// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath = @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

Example

The following example shows how to move files and directories.

```

// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}

```

Example

The following example shows how to delete files and directories.

```

// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

```

```
{  
    Console.WriteLine(e.Message);  
}  
// Delete a directory and all subdirectories with Directory static method...  
if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))  
{  
    try  
    {  
        System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);  
    }  
  
    catch (System.IO.IOException e)  
    {  
        Console.WriteLine(e.Message);  
    }  
}  
  
// ...or with DirectoryInfo instance method.  
System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");  
// Delete this dir and all subdirs.  
try  
{  
    di.Delete(true);  
}  
catch (System.IO.IOException e)  
{  
    Console.WriteLine(e.Message);  
}  
}  
}  
}
```

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)
- [How to provide a progress dialog box for file operations](#)
- [File and Stream I/O](#)
- [Common I/O Tasks](#)

How to provide a progress dialog box for file operations (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can provide a standard dialog box that shows progress on file operations in Windows if you use the `CopyFile(String, String, UIOption)` method in the `Microsoft.VisualBasic` namespace.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To add a reference in Visual Studio

1. On the menu bar, choose **Project, Add Reference**.
The **Reference Manager** dialog box appears.
2. In the **Assemblies** area, choose **Framework** if it isn't already chosen.
3. In the list of names, select the **Microsoft.VisualBasic** check box, and then choose the **OK** button to close the dialog box.

Example

The following code copies the directory that `sourcePath` specifies into the directory that `destinationPath` specifies. This code also provides a standard dialog box that shows the estimated amount of time remaining before the operation finishes.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.  
using Microsoft.VisualBasic.FileIO;  
  
class FileProgress  
{  
    static void Main()  
    {  
        // Specify the path to a folder that you want to copy. If the folder is small,  
        // you won't have time to see the progress dialog box.  
        string sourcePath = @"C:\Windows\symbols\";  
        // Choose a destination for the copied files.  
        string destinationPath = @"C:\TestFolder";  
  
        FileSystem.CopyDirectory(sourcePath, destinationPath,  
            UIOption.AllDialogs);  
    }  
}
```

See also

- [File System and the Registry \(C# Programming Guide\)](#)

How to write to a text file (C# Programming Guide)

3/6/2021 • 2 minutes to read • [Edit Online](#)

In this article, there are several examples showing various ways to write text to a file. The first two examples use static convenience methods on the [System.IO.File](#) class to write each element of any `IEnumerable<string>` and a `string` to a text file. The third example shows how to add text to a file when you have to process each line individually as you write to the file. In the first three examples, you overwrite all existing content in the file. The final example shows how to append text to an existing file.

These examples all write string literals to files. If you want to format text written to a file, use the [Format](#) method or C# [string interpolation](#) feature.

Write a collection of strings to a file

```
using System.IO;
using System.Threading.Tasks;

class WriteAllLines
{
    public static async Task ExampleAsync()
    {
        string[] lines =
        {
            "First line", "Second line", "Third line"
        };

        await File.WriteAllLinesAsync("WriteLines.txt", lines);
    }
}
```

The preceding source code example:

- Instantiates a string array with three values.
- Awaits a call to [File.WriteAllLinesAsync](#) which:
 - Asynchronously creates a file name *WriteLines.txt*. If the file already exists, it is overwritten.
 - Writes the given lines to the file.
 - Closes the file, automatically flushing and disposing as needed.

Write one string to a file

```

using System.IO;
using System.Threading.Tasks;

class WriteAllText
{
    public static async Task ExampleAsync()
    {
        string text =
            "A class is the most powerful data type in C#. Like a structure, " +
            "a class defines the data and behavior of the data type. ";

        await File.WriteAllTextAsync("WriteText.txt", text);
    }
}

```

The preceding source code example:

- Instantiates a string given the assigned string literal.
- Awaits a call to [File.WriteAllTextAsync](#) which:
 - Asynchronously creates a file name *WriteText.txt*. If the file already exists, it is overwritten.
 - Writes the given text to the file.
 - Closes the file, automatically flushing and disposing as needed.

Write selected strings from an array to a file

```

using System.IO;
using System.Threading.Tasks;

class StreamWriterOne
{
    public static async Task ExampleAsync()
    {
        string[] lines = { "First line", "Second line", "Third line" };
        using StreamWriter file = new("WriteLines2.txt");

        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                await file.WriteLineAsync(line);
            }
        }
    }
}

```

The preceding source code example:

- Instantiates a string array with three values.
- Instantiates a [StreamWriter](#) with a file path of *WriteLines2.txt* as a [using declaration](#).
- Iterates through all the lines.
- Conditionally awaits a call to [StreamWriter.WriteLineAsync\(String\)](#), which writes the line to the file when the line doesn't contain "Second".

Append text to an existing file

```
using System.IO;
using System.Threading.Tasks;

class StreamWriterTwo
{
    public static async Task ExampleAsync()
    {
        using StreamWriter file = new("WriteLines2.txt", append: true);
        await file.WriteLineAsync("Fourth line");
    }
}
```

The preceding source code example:

- Instantiates a string array with three values.
- Instantiates a [StreamWriter](#) with a file path of *WriteLines2.txt* as a [using declaration](#), passing in `true` to append.
- Awaits a call to [StreamWriter.WriteLineAsync\(String\)](#), which writes the string to the file as an appended line.

Exceptions

The following conditions may cause an exception:

- [InvalidOperationException](#): The file exists and is read-only.
- [PathTooLongException](#): The path name may be too long.
- [IOException](#): The disk may be full.

There are additional conditions that may cause exceptions when working with the file system, it is best to program defensively.

See also

- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)
- [Sample: Save a collection to Application Storage](#)

How to read from a text file (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example reads the contents of a text file by using the static methods `ReadAllText` and `ReadAllLines` from the `System.IO.File` class.

For an example that uses `StreamReader`, see [How to read a text file one line at a time](#).

NOTE

The files that are used in this example are created in the topic [How to write to a text file](#).

Example

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Compiling the Code

Copy the code and paste it into a C# console application.

If you are not using the text files from [How to write to a text file](#), replace the argument to `ReadAllText` and `ReadAllLines` with the appropriate path and file name on your computer.

Robust Programming

The following conditions may cause an exception:

- The file doesn't exist or doesn't exist at the specified location. Check the path and the spelling of the file name.

.NET Security

Do not rely on the name of a file to determine the contents of the file. For example, the file `myFile.cs` might not be a C# source file.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to read a text file one line at a time (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example reads the contents of a text file, one line at a time, into a string using the `ReadLine` method of the `StreamReader` class. Each text line is stored into the string `line` and displayed on the screen.

Example

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Compiling the Code

Copy the code and paste it into the `Main` method of a console application.

Replace `"c:\test.txt"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file may not exist.

.NET Security

Do not make decisions about the contents of the file based on the name of the file. For example, the file `myFile.cs` may not be a C# source file.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to create a key in the registry (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This example adds the value pair, "Name" and "Isabella", to the current user's registry, under the key "Names".

Example

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

Compiling the Code

- Copy the code and paste it into the `Main` method of a console application.
- Replace the `Names` parameter with the name of a key that exists directly under the `HKEY_CURRENT_USER` node of the registry.
- Replace the `Name` parameter with the name of a value that exists directly under the `Names` node.

Robust Programming

Examine the registry structure to find a suitable location for your key. For example, you might want to open the Software key of the current user, and create a key with your company's name. Then add the registry values to your company's key.

The following conditions might cause an exception:

- The name of the key is null.
- The user does not have permissions to create registry keys.
- The key name exceeds the 255-character limit.
- The key is closed.
- The registry key is read-only.

.NET Security

It is more secure to write data to the user folder — `Microsoft.Win32.Registry.CurrentUser` — rather than to the local computer — `Microsoft.Win32.Registry.LocalMachine`.

When you create a registry value, you need to decide what to do if that value already exists. Another process, perhaps a malicious one, may have already created the value and have access to it. When you put data in the registry value, the data is available to the other process. To prevent this, use the

`Overload:Microsoft.Win32.RegistryKey.GetValue` method. It returns null if the key does not already exist.

It is not secure to store secrets, such as passwords, in the registry as plain text, even if the registry key is protected by access control lists (ACL).

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)
- [Read, write and delete from the registry with C#](#)

Interoperability (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

[Interoperability Overview](#)

Describes methods to interoperate between C# managed code and unmanaged code.

[How to access Office interop objects by using C# features](#)

Describes features that are introduced in Visual C# to facilitate Office programming.

[How to use indexed properties in COM interop programming](#)

Describes how to use indexed properties to access COM properties that have parameters.

[How to use platform invoke to play a WAV file](#)

Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

[Walkthrough: Office Programming](#)

Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

[Example COM Class](#)

Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see [Basic concepts](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Marshal.ReleaseComObject](#)
- [C# Programming Guide](#)
- [Interoperating with Unmanaged Code](#)
- [Walkthrough: Office Programming](#)

Interoperability Overview (C# Programming Guide)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The topic describes methods to enable interoperability between C# managed code and unmanaged code.

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions that are implemented in dynamic link libraries (DLLs), such as those in the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see [Consuming Unmanaged DLL Functions](#) and [How to use platform invoke to play a WAV file](#).

NOTE

The [Common Language Runtime](#) (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see [Security in .NET](#).

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class so that it can be consumed by code that is authored in C# or another .NET language. To do this, you write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code to be located in the same application and even in the same file. You then build the C++ code by using the /clr compiler switch to produce a managed assembly. Finally, you add a reference to the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un-register a COM DLL.
2. Add to the project a reference to the COM component or type library.

When you add the reference, Visual Studio uses the [Tlbimp.exe \(Type Library Importer\)](#), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.

3. Create an instance of a class that is defined in the RCW. This, in turn, creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see [Exposing COM Components to the .NET Framework](#).

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project.

You can make an assembly COM visible by modifying Visual C# project properties. For more information, see [Assembly Information Dialog Box](#).

2. Generate a COM type library and register it for COM usage.

You can modify Visual C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the [Regasm.exe \(Assembly Registration Tool\)](#), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see [Exposing .NET Framework Components to COM](#) and [Example COM Class](#).

See also

- [Improving Interop Performance](#)
- [Introduction to Interoperability between COM and .NET](#)
- [Introduction to COM Interop in Visual Basic](#)
- [Marshaling between Managed and Unmanaged Code](#)
- [Interoperating with Unmanaged Code](#)
- [C# Programming Guide](#)

How to access Office interop objects (C# Programming Guide)

3/15/2021 • 12 minutes to read • [Edit Online](#)

C# has features that simplify access to Office API objects. The new features include named and optional arguments, a new type called `dynamic`, and the ability to pass arguments to reference parameters in COM methods as if they were value parameters.

In this topic you will use the new features to write code that creates and displays a Microsoft Office Excel worksheet. You will then write code to add an Office Word document that contains an icon that is linked to the Excel worksheet.

To complete this walkthrough, you must have Microsoft Office Excel 2007 and Microsoft Office Word 2007, or later versions, installed on your computer.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**. The **New Project** dialog box appears.
3. In the **Installed Templates** pane, expand **Visual C#**, and then click **Windows**.
4. Look at the top of the **New Project** dialog box to make sure that **.NET Framework 4** (or later version) is selected as a target framework.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **Assemblies** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Excel**. If you do not see the assemblies, you may need to ensure they are installed and displayed. See [How to: Install Office Primary Interop Assemblies](#).
3. Click **OK**.

To add necessary using directives

1. In Solution Explorer, right-click the `Program.cs` file and then click **View Code**.

2. Add the following `using` directives to the top of the code file:

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

To create a list of bank accounts

1. Paste the following class definition into `Program.cs`, under the `Program` class.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Add the following code to the `Main` method to create a `bankAccounts` list that contains two accounts.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
        ID = 345678,
        Balance = 541.27
    },
    new Account {
        ID = 1230221,
        Balance = -127.44
    }
};
```

To declare a method that exports account information to Excel

1. Add the following method to the `Program` class to set up an Excel worksheet.

Method `Add` has an optional parameter for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the following code, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument:

```
ExcelApp.Workbooks.Add(Type.Missing) .
```

```

static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}

```

2. Add the following code at the end of `DisplayInExcel`. The code inserts values into the first two columns of the first row of the worksheet.

```

// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";

```

3. Add the following code at the end of `DisplayInExcel`. The `foreach` loop puts the information from the list of accounts into the first two columns of successive rows of the worksheet.

```

var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}

```

4. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```

workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();

```

Earlier versions of C# require explicit casting for these operations because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel `Range` method. The following lines show the casting.

```

((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();

```

C# 4, and later versions, converts the returned `Object` to `dynamic` automatically if the assembly is referenced by the [EmbedInteropTypes](#) compiler option or, equivalently, if the Excel [Embed Interop Types](#) property is set to true. True is the default value for this property.

To run the project

- Add the following line at the end of `Main`.

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

2. Press CTRL+F5.

An Excel worksheet appears that contains the data from the two accounts.

To add a Word document

1. To illustrate additional ways in which C# 4, and later versions, enhances Office programming, the following code opens a Word application and creates an icon that links to the Excel worksheet.

Paste method `CreateIconInWordDoc`, provided later in this step, into the `Program` class.

`CreateIconInWordDoc` uses named and optional arguments to reduce the complexity of the method calls to `Add` and `PasteSpecial`. These calls incorporate two other new features introduced in C# 4 that simplify calls to COM methods that have reference parameters. First, you can send arguments to the reference parameters as if they were value parameters. That is, you can send values directly, without creating a variable for each reference parameter. The compiler generates temporary variables to hold the argument values, and discards the variables when you return from the call. Second, you can omit the `ref` keyword in the argument list.

The `Add` method has four reference parameters, all of which are optional. In C# 4.0 and later versions, you can omit arguments for any or all of the parameters if you want to use their default values. In C# 3.0 and earlier versions, an argument must be provided for each parameter, and the argument must be a variable because the parameters are reference parameters.

The `PasteSpecial` method inserts the contents of the Clipboard. The method has seven reference parameters, all of which are optional. The following code specifies arguments for two of them: `Link`, to create a link to the source of the Clipboard contents, and `DisplayAsIcon`, to display the link as an icon. In C# 4.0 and later versions, you can use named arguments for those two and omit the others. Although these are reference parameters, you do not have to use the `ref` keyword, or to create variables to send in as arguments. You can send the values directly. In C# 3.0 and earlier versions, you must supply a variable argument for each reference parameter.

```
static void CreateIconInWordDoc()  
{  
    var wordApp = new Word.Application();  
    wordApp.Visible = true;  
  
    // The Add method has four reference parameters, all of which are  
    // optional. Visual C# allows you to omit arguments for them if  
    // the default values are what you want.  
    wordApp.Documents.Add();  
  
    // PasteSpecial has seven reference parameters, all of which are  
    // optional. This example uses named arguments to specify values  
    // for two of the parameters. Although these are reference  
    // parameters, you do not need to use the ref keyword, or to create  
    // variables to send in as arguments. You can send the values directly.  
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);  
}
```

In C# 3.0 and earlier versions of the language, the following more complex code is required.

```

static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to construct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
        ref link,
        ref useDefaultValue,
        ref displayAsIcon,
        ref useDefaultValue,
        ref useDefaultValue,
        ref useDefaultValue);
}

```

2. Add the following statement at the end of `Main`.

```

// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();

```

3. Add the following statement at the end of `DisplayInExcel`. The `Copy` method adds the worksheet to the Clipboard.

```

// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();

```

4. Press **CTRL+F5**.

A Word document appears that contains an icon. Double-click the icon to bring the worksheet to the foreground.

To set the Embed Interop Types property

- Additional enhancements are possible when you call a COM type that does not require a primary interop assembly (PIA) at run time. Removing the dependency on PIAs results in version independence and easier deployment. For more information about the advantages of programming without PIAs, see [Walkthrough: Embedding Types from Managed Assemblies](#).

In addition, programming is easier because the types that are required and returned by COM methods

can be represented by using the type `dynamic` instead of `object`. Variables that have type `dynamic` are not evaluated until run time, which eliminates the need for explicit casting. For more information, see [Using Type dynamic](#).

In C# 4, embedding type information instead of using PIAs is default behavior. Because of that default, several of the previous examples are simplified because explicit casting is not required. For example, the declaration of `worksheet` in `DisplayInExcel` is written as

`Excel._Worksheet workSheet = excelApp.ActiveSheet` rather than

`Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. The calls to `AutoFit` in the same method also would require explicit casting without the default, because `ExcelApp.Columns[1]` returns an `object`, and `AutoFit` is an Excel method. The following code shows the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. To change the default and use PIAs instead of embedding type information, expand the **References** node in **Solution Explorer** and then select **Microsoft.Office.Interop.Excel** or **Microsoft.Office.Interop.Word**.
3. If you cannot see the **Properties** window, press **F4**.
4. Find **Embed Interop Types** in the list of properties, and change its value to **False**. Equivalently, you can compile by using the **References** compiler option instead of **EmbedInteropTypes** at a command prompt.

To add additional formatting to the table

1. Replace the two calls to `AutoFit` in `DisplayInExcel` with the following statement.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

The `AutoFormat` method has seven value parameters, all of which are optional. Named and optional arguments enable you to provide arguments for none, some, or all of them. In the previous statement, an argument is supplied for only one of the parameters, `Format`. Because `Format` is the first parameter in the parameter list, you do not have to provide the parameter name. However, the statement might be easier to understand if the parameter name is included, as is shown in the following code.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Press **CTRL+F5** to see the result. Other formats are listed in the **XlRangeAutoFormat** enumeration.
3. Compare the statement in step 1 with the following code, which shows the arguments that are required in C# 3.0 and earlier versions.

```

// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);

```

Example

The following code shows the complete example.

```

using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                    ID = 345678,
                    Balance = 541.27
                },
                new Account {
                    ID = 1230221,
                    Balance = -127.44
                }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a particular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;

```

```

// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";

var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}

workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();

// Call to AutoFormat in Visual C#. This statement replaces the
// two calls to AutoFit.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
}

static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
}
}

public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
}

```

See also

- [Type.Missing](#)
- [dynamic](#)
- [Using Type dynamic](#)
- [Named and Optional Arguments](#)
- [How to use named and optional arguments in Office programming](#)

How to use indexed properties in COM interop programming (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Indexed properties improve the way in which COM properties that have parameters are consumed in C# programming. Indexed properties work together with other features in Visual C#, such as [named and optional arguments](#), a new type ([dynamic](#)), and [embedded type information](#), to enhance Microsoft Office programming.

In earlier versions of C#, methods are accessible as properties only if the `get` method has no parameters and the `set` method has one and only one value parameter. However, not all COM properties meet those restrictions. For example, the Excel `Range[]` property has a `get` accessor that requires a parameter for the name of the range. In the past, because you could not access the `Range` property directly, you had to use the `get_Range` method instead, as shown in the following example.

```
// Visual C# 2008 and earlier.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Indexed properties enable you to write the following instead:

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

The previous example also uses the [optional arguments](#) feature, which enables you to omit `Type.Missing`.

Similarly to set the value of the `Value` property of a `Range` object in C# 3.0 and earlier, two arguments are required. One supplies an argument for an optional parameter that specifies the type of the range value. The other supplies the value for the `Value` property. The following examples illustrate these techniques. Both set the value of the A1 cell to `Name`.

```
// Visual C# 2008.  
targetRange.set_Value(Type.Missing, "Name");  
// Or  
targetRange.Value2 = "Name";
```

Indexed properties enable you to write the following code instead.

```
// Visual C# 2010.  
targetRange.Value = "Name";
```

You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

Example

The following code shows a complete example. For more information about how to set up a project that accesses the Office API, see [How to access Office interop objects by using C# features](#).

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

See also

- [Named and Optional Arguments](#)
- [dynamic](#)
- [Using Type dynamic](#)
- [How to use named and optional arguments in Office programming](#)
- [How to access Office interop objects by using C# features](#)
- [Walkthrough: Office Programming](#)

How to use platform invoke to play a WAV file (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following C# code example illustrates how to use platform invoke services to play a WAV sound file on the Windows operating system.

Example

This example code uses `DllImportAttribute` to import `winmm.dll`'s `PlaySound` method entry point as `Form1.PlaySound()`. The example has a simple Windows Form with a button. Clicking the button opens a standard windows `OpenFileDialog` dialog box so that you can open a file to play. When a wave file is selected, it is played by using the `PlaySound()` method of the `winmm.dll` library. For more information about this method, see [Using the PlaySound function with Waveform-Audio Files](#). Browse and select a file that has a .wav extension, and then click **Open** to play the wave file by using platform invoke. A text box shows the full path of the file selected.

The **Open Files** dialog box is filtered to show only files that have a .wav extension through the filter settings:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```

using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1() // Constructor.
        {
            InitializeComponent();
        }

        [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true, CharSet = CharSet.Unicode,
        ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            var dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if (dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound(dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Including this empty method in the sample because in the IDE,
            // when users click on the form, generates code that looks for a default method
            // with this name. We add it here to prevent confusion for those using the samples.
        }
    }
}

```

Compiling the code

1. Create a new C# Windows Forms Application project in Visual Studio and name it **WinSound**.
2. Copy the code above, and paste it over the contents of the *Form1.cs* file.
3. Copy the following code, and paste it in the *Form1.Designer.cs* file, in the `InitializeComponent()` method,

after any existing code.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Compile and run the code.

See also

- [C# Programming Guide](#)
- [Interoperability Overview](#)
- [A Closer Look at Platform Invoke](#)
- [Marshaling Data with Platform Invoke](#)

Walkthrough: Office Programming (C# and Visual Basic)

3/15/2021 • 11 minutes to read • [Edit Online](#)

Visual Studio offers features in C# and Visual Basic that improve Microsoft Office programming. Helpful C# features include named and optional arguments and return values of type `dynamic`. In COM programming, you can omit the `ref` keyword and gain access to indexed properties. Features in Visual Basic include auto-implemented properties, statements in lambda expressions, and collection initializers.

Both languages enable embedding of type information, which allows deployment of assemblies that interact with COM components without deploying primary interop assemblies (PIAs) to the user's computer. For more information, see [Walkthrough: Embedding Types from Managed Assemblies](#).

This walkthrough demonstrates these features in the context of Office programming, but many of these features are also useful in general programming. In the walkthrough, you use an Excel Add-in application to create an Excel workbook. Next, you create a Word document that contains a link to the workbook. Finally, you see how to enable and disable the PIA dependency.

Prerequisites

You must have Microsoft Office Excel and Microsoft Office Word installed on your computer to complete this walkthrough.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To set up an Excel Add-in application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Installed Templates** pane, expand **Visual Basic** or **Visual C#**, expand **Office**, and then click the version year of the Office product.
4. In the **Templates** pane, click **Excel <version> Add-in**.
5. Look at the top of the **Templates** pane to make sure that **.NET Framework 4**, or a later version, appears in the **Target Framework** box.
6. Type a name for your project in the **Name** box, if you want to.
7. Click **OK**.
8. The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.

2. On the **Assemblies** tab, select **Microsoft.Office.Interop.Excel**, version <version>.0.0.0 (for a key to the Office product version numbers, see [Microsoft Versions](#)), in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Word**, version <version>.0.0.0. If you do not see the assemblies, you may need to ensure they are installed and displayed (see [How to: Install Office Primary Interop Assemblies](#)).

3. Click **OK**.

To add necessary Imports statements or using directives

1. In **Solution Explorer**, right-click the **ThisAddIn.vb** or **ThisAddIn.cs** file and then click **View Code**.
2. Add the following **Imports** statements (Visual Basic) or **using** directives (C#) to the top of the code file if they are not already present.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

To create a list of bank accounts

1. In **Solution Explorer**, right-click your project's name, click **Add**, and then click **Class**. Name the class **Account.vb** if you are using Visual Basic or **Account.cs** if you are using C#. Click **Add**.
2. Replace the definition of the **Account** class with the following code. The class definitions use *auto-implemented properties*. For more information, see [Auto-Implemented Properties](#).

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. To create a **bankAccounts** list that contains two accounts, add the following code to the **ThisAddIn_Startup** method in **ThisAddIn.vb** or **ThisAddIn.cs**. The list declarations use *collection initializers*. For more information, see [Collection Initializers](#).

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```

Dim bankAccounts As New List(Of Account) From {
    New Account With {
        .ID = 345,
        .Balance = 541.27
    },
    New Account With {
        .ID = 123,
        .Balance = -127.44
    }
}

```

To export data to Excel

1. In the same file, add the following method to the `ThisAddIn` class. The method sets up an Excel workbook and exports data to it.

```

void DisplayInExcel(IEnumerable<Account> accounts,
                    Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}

```

```

Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub

```

Two new C# features are used in this method. Both of these features already exist in Visual Basic.

- Method `Add` has an *optional parameter* for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the previous example, `Add` uses the

default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `excelApp.Workbooks.Add(Type.Missing)`.

For more information, see [Named and Optional Arguments](#).

- The `Range` and `offset` properties of the `Range` object use the *indexed properties* feature. This feature enables you to consume these properties from COM types by using the following typical C# syntax. Indexed properties also enable you to use the `Value` property of the `Range` object, eliminating the need to use the `Value2` property. The `Value` property is indexed, but the index is optional. Optional arguments and indexed properties work together in the following example.

```
// Visual C# 2010 provides indexed properties for COM programming.  
excelApp.Range["A1"].Value = "ID";  
excelApp.ActiveCell.Offset[1, 0].Select();
```

In earlier versions of the language, the following special syntax is required.

```
// In Visual C# 2008, you cannot access the Range, Offset, and Value  
// properties directly.  
excelApp.get_Range("A1").Value2 = "ID";  
excelApp.ActiveCell.get_Offset(1, 0).Select();
```

You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

For more information, see [How to use indexed properties in COM interop programming](#).

2. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
excelApp.Columns[1].AutoFit();  
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.  
.Columns(1).AutoFit()  
.Columns(2).AutoFit()
```

These additions demonstrate another feature in C#: treating `Object` values returned from COM hosts such as Office as if they have type `dynamic`. This happens automatically when **Embed Interop Types** is set to its default value, `True`, or, equivalently, when the assembly is referenced by the **EmbedInteropTypes** compiler option. Type `dynamic` allows late binding, already available in Visual Basic, and avoids the explicit casting required in C# 3.0 and earlier versions of the language.

For example, `excelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel `Range` method. Without `dynamic`, you must cast the object returned by `excelApp.Columns[1]` as an instance of `Range` before calling method `AutoFit`.

```
// Casting is required in Visual C# 2008.  
((Excel.Range)excelApp.Columns[1]).AutoFit();  
  
// Casting is not required in Visual C# 2010.  
excelApp.Columns[1].AutoFit();
```

For more information about embedding interop types, see procedures "To find the PIA reference" and "To restore the PIA dependency" later in this topic. For more information about `dynamic`, see [dynamic](#) or

Using Type dynamic.

To invoke DisplayInExcel

1. Add the following code at the end of the `ThisAddIn_StartUp` method. The call to `DisplayInExcel` contains two arguments. The first argument is the name of the list of accounts to be processed. The second argument is a multiline lambda expression that defines how the data is to be processed. The `ID` and `balance` values for each account are displayed in adjacent cells, and the row is displayed in red if the balance is less than zero. For more information, see [Lambda Expressions](#).

```
DisplayInExcel(bankAccounts, (account, cell) =>
    // This multiline lambda expression sets custom processing rules
    // for the bankAccounts.
    {
        cell.Value = account.ID;
        cell.Offset[0, 1].Value = account.Balance;
        if (account.Balance < 0)
        {
            cell.Interior.Color = 255;
            cell.Offset[0, 1].Interior.Color = 255;
        }
    });
}
```

```
DisplayInExcel(bankAccounts,
    Sub(account, cell)
        ' This multiline lambda expression sets custom
        ' processing rules for the bankAccounts.
        cell.Value = account.ID
        cell.Offset(0, 1).Value = account.Balance

        If account.Balance < 0 Then
            cell.Interior.Color = RGB(255, 0, 0)
            cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
        End If
    End Sub)
```

2. To run the program, press F5. An Excel worksheet appears that contains the data from the accounts.

To add a Word document

1. Add the following code at the end of the `ThisAddIn_StartUp` method to create a Word document that contains a link to the Excel workbook.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

This code demonstrates several of the new features in C#: the ability to omit the `ref` keyword in COM programming, named arguments, and optional arguments. These features already exist in Visual Basic. The `PasteSpecial` method has seven parameters, all of which are defined as optional reference parameters. Named and optional arguments enable you to designate the parameters you want to access by name and to send arguments to only those parameters. In this example, arguments are sent to

indicate that a link to the workbook on the Clipboard should be created (parameter `Link`) and that the link is to be displayed in the Word document as an icon (parameter `DisplayAsIcon`). Visual C# also enables you to omit the `ref` keyword for these arguments.

To run the application

1. Press F5 to run the application. Excel starts and displays a table that contains the information from the two accounts in `bankAccounts`. Then a Word document appears that contains a link to the Excel table.

To clean up the completed project

1. In Visual Studio, click **Clean Solution** on the **Build** menu. Otherwise, the add-in will run every time that you open Excel on your computer.

To find the PIA reference

1. Run the application again, but do not click **Clean Solution**.
2. Select the **Start**. Locate **Microsoft Visual Studio <version>** and open a developer command prompt.
3. Type `ildasm` in the Developer Command Prompt for Visual Studio window, and then press ENTER. The IL DASM window appears.
4. On the **File** menu in the IL DASM window, select **File > Open**. Double-click **Visual Studio <version>**, and then double-click **Projects**. Open the folder for your project, and look in the bin/Debug folder for *your project name.dll*. Double-click *your project name.dll*. A new window displays your project's attributes, in addition to references to other modules and assemblies. Note that namespaces `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are included in the assembly. By default in Visual Studio, the compiler imports the types you need from a referenced PIA into your assembly.

For more information, see [How to: View Assembly Contents](#).

5. Double-click the **MANIFEST** icon. A window appears that contains a list of assemblies that contain items referenced by the project. `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are not included in the list. Because the types your project needs have been imported into your assembly, references to a PIA are not required. This makes deployment easier. The PIAs do not have to be present on the user's computer, and because an application does not require deployment of a specific version of a PIA, applications can be designed to work with multiple versions of Office, provided that the necessary APIs exist in all versions.

Because deployment of PIAs is no longer necessary, you can create an application in advanced scenarios that works with multiple versions of Office, including earlier versions. However, this works only if your code does not use any APIs that are not available in the version of Office you are working with. It is not always clear whether a particular API was available in an earlier version, and for that reason working with earlier versions of Office is not recommended.

NOTE

Office did not publish PIAs before Office 2003. Therefore, the only way to generate an interop assembly for Office 2002 or earlier versions is by importing the COM reference.

6. Close the manifest window and the assembly window.

To restore the PIA dependency

1. In **Solution Explorer**, click the **Show All Files** button. Expand the **References** folder and select `Microsoft.Office.Interop.Excel`. Press F4 to display the **Properties** window.
2. In the **Properties** window, change the **Embed Interop Types** property from **True** to **False**.

3. Repeat steps 1 and 2 in this procedure for `Microsoft.Office.Interop.Word`.
4. In C#, comment out the two calls to `Autofit` at the end of the `DisplayInExcel` method.
5. Press F5 to verify that the project still runs correctly.
6. Repeat steps 1-3 from the previous procedure to open the assembly window. Notice that `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are no longer in the list of embedded assemblies.
7. Double-click the **MANIFEST** icon and scroll through the list of referenced assemblies. Both `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are in the list. Because the application references the Excel and Word PIAs, and the **Embed Interop Types** property is set to **False**, both assemblies must exist on the end user's computer.
8. In Visual Studio, click **Clean Solution** on the **Build** menu to clean up the completed project.

See also

- [Auto-Implemented Properties \(Visual Basic\)](#)
- [Auto-Implemented Properties \(C#\)](#)
- [Collection Initializers](#)
- [Object and Collection Initializers](#)
- [Optional Parameters](#)
- [Passing Arguments by Position and by Name](#)
- [Named and Optional Arguments](#)
- [Early and Late Binding](#)
- [dynamic](#)
- [Using Type dynamic](#)
- [Lambda Expressions \(Visual Basic\)](#)
- [Lambda Expressions \(C#\)](#)
- [How to use indexed properties in COM interop programming](#)
- [Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio](#)
- [Walkthrough: Embedding Types from Managed Assemblies](#)
- [Walkthrough: Creating Your First VSTO Add-in for Excel](#)
- [COM Interop](#)
- [Interoperability](#)

Example COM Class (C# Programming Guide)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following is an example of a class that you would expose as a COM object. After this code has been placed in a .cs file and added to your project, set the **Register for COM Interop** property to **True**. For more information, see [How to: Register a Component for COM Interop](#).

Exposing Visual C# objects to COM requires declaring a class interface, an events interface if it is required, and the class itself. Class members must follow these rules to be visible to COM:

- The class must be public.
- Properties, methods, and events must be public.
- Properties and methods must be declared on the class interface.
- Events must be declared in the event interface.

Other public members in the class that are not declared in these interfaces will not be visible to COM, but they will be visible to other .NET objects.

To expose properties and methods to COM, you must declare them on the class interface and mark them with a `DispId` attribute, and implement them in the class. The order in which the members are declared in the interface is the order used for the COM vtable.

To expose events from your class, you must declare them on the events interface and mark them with a `DispId` attribute. The class should not implement this interface.

The class implements the class interface; it can implement more than one interface, but the first implementation will be the default class interface. Implement the methods and properties exposed to COM here. They must be marked public and must match the declarations in the class interface. Also, declare the events raised by the class here. They must be marked public and must match the declarations in the events interface.

Example

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
     InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
     ClassInterface(ClassInterfaceType.None),
     ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

See also

- [C# Programming Guide](#)
- [Interoperability](#)
- [Build Page, Project Designer \(C#\)](#)

C# reference

3/27/2021 • 3 minutes to read • [Edit Online](#)

This section provides reference material about C# keywords, operators, special characters, preprocessor directives, compiler options, and compiler errors and warnings.

In this section

[C# Keywords](#)

Provides links to information about C# keywords and syntax.

[C# Operators](#)

Provides links to information about C# operators and syntax.

[C# Special Characters](#)

Provides links to information about special contextual characters in C# and their usage.

[C# Preprocessor Directives](#)

Provides links to information about compiler commands for embedding in C# source code.

[C# Compiler Options](#)

Includes information about compiler options and how to use them.

[C# Compiler Errors](#)

Includes code snippets that demonstrate the cause and correction of C# compiler errors and warnings.

[C# Language Specification](#)

The C# 6.0 language specification. This is a draft proposal for the C# 6.0 language. This document will be refined through work with the ECMA C# standards committee. Version 5.0 has been released in December 2017 as the [Standard ECMA-334 5th Edition](#) document.

The features that have been implemented in C# versions after 6.0 are represented in language specification proposals. These documents describe the deltas to the language spec in order to add these new features. These are in draft proposal form. These specifications will be refined and submitted to the ECMA standards committee for formal review and incorporation into a future version of the C# Standard.

[C# 7.0 Specification Proposals](#)

There are a number of new features implemented in C# 7.0. They include pattern matching, local functions, out variable declarations, throw expressions, binary literals, and digit separators. This folder contains the specifications for each of those features.

[C# 7.1 Specification Proposals](#)

There are new features added in C# 7.1. First, you can write a `Main` method that returns `Task` or `Task<int>`. This enables you to add the `async` modifier to `Main`. The `default` expression can be used without a type in locations where the type can be inferred. Also, tuple member names can be inferred. Finally, pattern matching can be used with generics.

[C# 7.2 Specification Proposals](#)

C# 7.2 added a number of small features. You can pass arguments by readonly reference using the `in` keyword. There are a number of low-level changes to support compile-time safety for `Span` and related types. You can use named arguments where later arguments are positional, in some situations. The `private protected` access modifier enables you to specify that callers are limited to derived types implemented in the same assembly. The `?:` operator can resolve to a reference to a variable. You can also format hexadecimal and binary numbers

using a leading digit separator.

C# 7.3 Specification Proposals

C# 7.3 is another point release that includes several small updates. You can use new constraints on generic type parameters. Other changes make it easier to work with `fixed` fields, including using `stackalloc` allocations. Local variables declared with the `ref` keyword may be reassigned to refer to new storage. You can place attributes on auto-implemented properties that target the compiler-generated backing field. Expression variables can be used in initializers. Tuples can be compared for equality (or inequality). There have also been some improvements to overload resolution.

C# 8.0 Specification Proposals

C# 8.0 is available with .NET Core 3.0. The features include nullable reference types, recursive pattern matching, default interface methods, async streams, ranges and indexes, pattern based using and using declarations, null coalescing assignment, and readonly instance members.

C# 9.0 Specification Proposals

C# 9.0 is available with .NET 5.0. The features include records, top-level statements, pattern matching enhancements, init only setters, target-typed new expressions, module initializers, extending partial methods, static anonymous functions, target-typed conditional expressions, covariant return types, extension `GetEnumerator` in foreach loops, lambda discard parameters, attributes on local functions, native sized integers, function pointers, suppress emitting `localsinit` flag, and unconstrained type parameter annotations.

Related sections

[Using the Visual Studio Development Environment for C#](#)

Provides links to conceptual and task topics that describe the IDE and Editor.

[C# Programming Guide](#)

Includes information about how to use the C# programming language.

C# language versioning

3/27/2021 • 4 minutes to read • [Edit Online](#)

The latest C# compiler determines a default language version based on your project's target framework or frameworks. Visual Studio doesn't provide a UI to change the value, but you can change it by editing the `csproj` file. The choice of default ensures that you use the latest language version compatible with your target framework. You benefit from access to the latest language features compatible with your project's target. This default choice also ensures you don't use a language that requires types or runtime behavior not available in your target framework. Choosing a language version newer than the default can cause hard to diagnose compile-time and runtime errors.

The rules in this article apply to the compiler delivered with Visual Studio 2019 or the .NET SDK. The C# compilers that are part of the Visual Studio 2017 installation or earlier .NET Core SDK versions target C# 7.0 by default.

C# 8.0 is supported only on .NET Core 3.x and newer versions. Many of the newest features require library and runtime features introduced in .NET Core 3.x:

- [Default interface implementation](#) requires new features in the .NET Core 3.0 CLR.
- [Async streams](#) require the new types `System.IAsyncDisposable`, `System.Collections.Generic.IAsyncEnumerable<T>`, and `System.Collections.Generic.IAsyncEnumerator<T>`.
- [Indices and ranges](#) require the new types `System.Index` and `System.Range`.
- [Nullable reference types](#) make use of several [attributes](#) to provide better warnings. Those attributes were added in .NET Core 3.0. Other target frameworks haven't been annotated with any of these attributes. That means nullable warnings may not accurately reflect potential issues.

C# 9.0 is supported only on .NET 5 and newer versions.

Defaults

The compiler determines a default based on these rules:

TARGET FRAMEWORK	VERSION	C# LANGUAGE VERSION DEFAULT
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

When your project targets a preview framework that has a corresponding preview language version, the language version used is the preview language version. You use the latest features with that preview in any

environment, without affecting projects that target a released .NET Core version.

IMPORTANT

Visual Studio 2017 added a `<LangVersion>latest</LangVersion>` entry to any project files it created. That meant C# 7.0 when it was added. However, once you upgrade to Visual Studio 2019, that means the latest released version, regardless of the target framework. These projects now [override the default behavior](#). You should edit the project file and remove that node. Then, your project will use the compiler version recommended for your target framework.

Override a default

If you must specify your C# version explicitly, you can do so in several ways:

- Manually edit your [project file](#).
- Set the language version [for multiple projects in a subdirectory](#).
- Configure the [LangVersion compiler option](#).

TIP

To know what language version you're currently using, put `#error version` (case sensitive) in your code. This makes the compiler report a compiler error, CS8304, with a message containing the compiler version being used and the current selected language version. See [#error \(C# Reference\)](#) for more information.

Edit the project file

You can set the language version in your project file. For example, if you explicitly want access to preview features, add an element like this:

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

The value `preview` uses the latest available preview C# language version that your compiler supports.

Configure multiple projects

To configure multiple projects, you can create a `Directory.Build.props` file that contains the `<LangVersion>` element. You typically do that in your solution directory. Add the following to a `Directory.Build.props` file in your solution directory:

```
<Project>
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

Builds in all subdirectories of the directory containing that file will use the preview C# version. For more information, see [Customize your build](#).

C# language version reference

The following table shows all current C# language versions. Your compiler may not necessarily understand every value if it's older. If you install the latest .NET SDK, then you have access to everything listed.

VALUE	MEANING
<code>preview</code>	The compiler accepts all valid language syntax from the latest preview version.
<code>latest</code>	The compiler accepts syntax from the latest released version of the compiler (including minor version).
<code>latestMajor</code> (default)	The compiler accepts syntax from the latest released major version of the compiler.
<code>9.0</code>	The compiler accepts only syntax that is included in C# 9.0 or lower.
<code>8.0</code>	The compiler accepts only syntax that is included in C# 8.0 or lower.
<code>7.3</code>	The compiler accepts only syntax that is included in C# 7.3 or lower.
<code>7.2</code>	The compiler accepts only syntax that is included in C# 7.2 or lower.
<code>7.1</code>	The compiler accepts only syntax that is included in C# 7.1 or lower.
<code>7</code>	The compiler accepts only syntax that is included in C# 7.0 or lower.
<code>6</code>	The compiler accepts only syntax that is included in C# 6.0 or lower.
<code>5</code>	The compiler accepts only syntax that is included in C# 5.0 or lower.
<code>4</code>	The compiler accepts only syntax that is included in C# 4.0 or lower.
<code>3</code>	The compiler accepts only syntax that is included in C# 3.0 or lower.
<code>ISO-2</code> (or <code>2</code>)	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
<code>ISO-1</code> (or <code>1</code>)	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

TIP

Open [Visual Studio Developer Command Prompt](#) or [Visual Studio Developer PowerShell](#), and run the following command to see the listing of language versions available on your machine.

```
csc -langversion:?
```

Querying the `**LangVersion` compile option like this prints something similar to the following:

```
Supported language versions:  
default  
1  
2  
3  
4  
5  
6  
7.0  
7.1  
7.2  
7.3  
8.0  
9.0 (default)  
latestmajor  
preview  
latest
```

Value types (C# reference)

3/6/2021 • 3 minutes to read • [Edit Online](#)

Value types and [reference types](#) are the two main categories of C# types. A variable of a value type contains an instance of the type. This differs from a variable of a reference type, which contains a reference to an instance of the type. By default, on [assignment](#), passing an argument to a method, and returning a method result, variable values are copied. In the case of value-type variables, the corresponding type instances are copied. The following example demonstrates that behavior:

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

As the preceding example shows, operations on a value-type variable affect only that instance of the value type, stored in the variable.

If a value type contains a data member of a reference type, only the reference to the instance of the reference type is copied when a value-type instance is copied. Both the copy and original value-type instance have access to the same reference-type instance. The following example demonstrates that behavior:

```

using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1); // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1); // output: 0 [A, B]
        Console.WriteLine(n2); // output: 7 [A, B]
    }
}

```

NOTE

To make your code less error-prone and more robust, define and use immutable value types. This article uses mutable value types only for demonstration purposes.

Kinds of value types and type constraints

A value type can be one of the two following kinds:

- a [structure type](#), which encapsulates data and related functionality
- an [enumeration type](#), which is defined by a set of named constants and represents a choice or a combination of choices

A [nullable value type](#) `T?` represents all values of its underlying value type `T` and an additional `null` value. You cannot assign `null` to a variable of a value type, unless it's a nullable value type.

You can use the `struct constraint` to specify that a type parameter is a non-nullable value type. Both structure and enumeration types satisfy the `struct constraint`. Beginning with C# 7.3, you can use `System.Enum` in a base class constraint (that is known as the [enum constraint](#)) to specify that a type parameter is an enumeration type.

Built-in value types

C# provides the following built-in value types, also known as *simple types*:

- [Integral numeric types](#)
- [Floating-point numeric types](#)
- `bool` that represents a Boolean value
- `char` that represents a Unicode UTF-16 character

All simple types are structure types and differ from other structure types in that they permit certain additional operations:

- You can use literals to provide a value of a simple type. For example, `'A'` is a literal of the type `char` and `2001` is a literal of the type `int`.
- You can declare constants of the simple types with the `const` keyword. It's not possible to have constants of other structure types.
- Constant expressions, whose operands are all constants of the simple types, are evaluated at compile time.

Beginning with C# 7.0, C# supports [value tuples](#). A value tuple is a value type, but not a simple type.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Value types](#)
- [Simple types](#)
- [Variables](#)

See also

- [C# reference](#)
- [System.ValueType](#)
- [Reference types](#)

Integral numeric types (C# reference)

3/20/2021 • 3 minutes to read • [Edit Online](#)

The *integral numeric types* represent integer numbers. All integral numeric types are [value types](#). They are also [simple types](#) and can be initialized with [literals](#). All integral numeric types support [arithmetic](#), [bitwise logical](#), [comparison](#), and [equality](#) operators.

Characteristics of the integral types

C# supports the following predefined integral types:

C# TYPE/KEYWORD	RANGE	SIZE	.NET TYPE
<code>sbyte</code>	-128 to 127	Signed 8-bit integer	System.SByte
<code>byte</code>	0 to 255	Unsigned 8-bit integer	System.Byte
<code>short</code>	-32,768 to 32,767	Signed 16-bit integer	System.Int16
<code>ushort</code>	0 to 65,535	Unsigned 16-bit integer	System.UInt16
<code>int</code>	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
<code>uint</code>	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
<code>long</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
<code>ulong</code>	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
<code>nint</code>	Depends on platform	Signed 32-bit or 64-bit integer	System.IntPtr
<code>nuint</code>	Depends on platform	Unsigned 32-bit or 64-bit integer	System.UIntPtr

In all of the table rows except the last two, each C# type keyword from the leftmost column is an alias for the corresponding .NET type. The keyword and .NET type name are interchangeable. For example, the following declarations declare variables of the same type:

```
int a = 123;
System.Int32 b = 123;
```

The `nint` and `nuint` types in the last two rows of the table are native-sized integers. They are represented internally by the indicated .NET types, but in each case the keyword and the .NET type are not interchangeable.

The compiler provides operations and conversions for `nint` and `nuint` as integer types that it doesn't provide for the pointer types `System.IntPtr` and `System.UIntPtr`. For more information, see [nint and nuint types](#).

For information about the native-sized integer types, see [nint and nuint](#).

The default value of each integral type is zero, `0`. Each of the integral types except the native-sized types has `MinValue` and `MaxValue` constants that provide the minimum and maximum value of that type.

Use the `System.Numerics.BigInteger` structure to represent a signed integer with no upper or lower bounds.

Integer literals

Integer literals can be

- *decimal*: without any prefix
- *hexadecimal*: with the `0x` or `0X` prefix
- *binary*: with the `0b` or `0B` prefix (available in C# 7.0 and later)

The following code demonstrates an example of each:

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

The preceding example also shows the use of `_` as a *digit separator*, which is supported starting with C# 7.0. You can use the digit separator with all kinds of numeric literals.

The type of an integer literal is determined by its suffix as follows:

- If the literal has no suffix, its type is the first of the following types in which its value can be represented: `int`, `uint`, `long`, `ulong`.
- If the literal is suffixed by `u` or `U`, its type is the first of the following types in which its value can be represented: `uint`, `ulong`.
- If the literal is suffixed by `L` or `l`, its type is the first of the following types in which its value can be represented: `long`, `ulong`.

NOTE

You can use the lowercase letter `l` as a suffix. However, this generates a compiler warning because the letter `l` can be confused with the digit `1`. Use `L` for clarity.

- If the literal is suffixed by `UL`, `U1`, `uL`, `u1`, `LU`, `Lu`, `lu`, or `lu`, its type is `ulong`.

If the value represented by an integer literal exceeds `UInt64.MaxValue`, a compiler error [CS1021](#) occurs.

If the determined type of an integer literal is `int` and the value represented by the literal is within the range of the destination type, the value can be implicitly converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` or `nuint`:

```
byte a = 17;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a 'byte'
```

As the preceding example shows, if the literal's value is not within the range of the destination type, a compiler error [CS0031](#) occurs.

You can also use a cast to convert the value represented by an integer literal to the type other than the determined type of the literal:

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Conversions

You can convert any integral numeric type to any other integral numeric type. If the destination type can store all values of the source type, the conversion is implicit. Otherwise, you need to use a [cast expression](#) to perform an explicit conversion. For more information, see [Built-in numeric conversions](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Integral types](#)
- [Integer literals](#)

See also

- [C# reference](#)
- [Value types](#)
- [Floating-point types](#)
- [Standard numeric format strings](#)
- [Numerics in .NET](#)

`nint` and `nuint` types (C# reference)

3/20/2021 • 2 minutes to read • [Edit Online](#)

Starting in C# 9.0, you can use the `nint` and `nuint` keywords to define *native-sized integers*. These are 32-bit integers when running in a 32-bit process, or 64-bit integers when running in a 64-bit process. They can be used for interop scenarios, low-level libraries, and to optimize performance in scenarios where integer math is used extensively.

The native-sized integer types are represented internally as the .NET types `System.IntPtr` and `System.UIntPtr`. Unlike other numeric types, the keywords are not simply aliases for the types. The following statements are not equivalent:

```
nint a = 1;
System.IntPtr a = 1;
```

The compiler provides operations and conversions for `nint` and `nuint` that are appropriate for integer types.

Run-time native integer size

To get the size of a native-sized integer at run time, you can use `sizeof()`. However, the code must be compiled in an unsafe context. For example:

```
Console.WriteLine($"size of nint = {sizeof(nint)}");
Console.WriteLine($"size of nuint = {sizeof(nuint)}");

// output when run in a 64-bit process
//size of nint = 8
//size of nuint = 8

// output when run in a 32-bit process
//size of nint = 4
//size of nuint = 4
```

You can also get the equivalent value from the static `IntPtr.Size` and `UIntPtr.Size` properties.

MinValue and MaxValue

To get the minimum and maximum values of native-sized integers at run time, use `MinValue` and `MaxValue` as static properties with the `nint` and `nuint` keywords, as in the following example:

```
Console.WriteLine($"nint.MinValue = {nint.MinValue}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

// output when run in a 64-bit process
//nint.MinValue = -9223372036854775808
//nint.MaxValue = 9223372036854775807
//nuint.MinValue = 0
//nuint.MaxValue = 18446744073709551615

// output when run in a 32-bit process
//nint.MinValue = -2147483648
//nint.MaxValue = 2147483647
//nuint.MinValue = 0
//nuint.MaxValue = 4294967295
```

Constants

You can use constant values in the following ranges:

- For `nint` : [Int32.MinValue](#) to [Int32.MaxValue](#).
- For `nuint` : [UInt32.MinValue](#) to [UInt32.MaxValue](#).

Conversions

The compiler provides implicit and explicit conversions to other numeric types. For more information, see [Built-in numeric conversions](#).

Literals

There's no direct syntax for native-sized integer literals. There's no suffix to indicate that a literal is a native-sized integer, such as `L` to indicate a `long`. You can use implicit or explicit casts of other integer values instead. For example:

```
nint a = 42
nint a = (nint)42;
```

Unsupported IntPtr/UIntPtr members

The following members of [IntPtr](#) and [UIntPtr](#) aren't supported for `nint` and `nuint` types:

- Parameterized constructors
- [Add\(IntPtr, Int32\)](#)
- [CompareTo](#)
- [Size](#) - Use [sizeOf\(\)](#) instead. Although `nint.size` isn't supported, you can use `IntPtr.Size` to get an equivalent value.
- [Subtract\(IntPtr, Int32\)](#)
- [ToInt32](#)
- [ToInt64](#)
- [ToPointer](#)
- [Zero](#) - Use 0 instead.

C# language specification

For more information, see the [C# language specification](#) and the [Native-sized integers](#) section of the C# 9.0 feature proposal notes.

See also

- [C# reference](#)
- [Value types](#)
- [Integral numeric types](#)
- [Built-in numeric conversions](#)

Floating-point numeric types (C# reference)

3/6/2021 • 4 minutes to read • [Edit Online](#)

The *floating-point numeric types* represent real numbers. All floating-point numeric types are [value types](#). They are also [simple types](#) and can be initialized with [literals](#). All floating-point numeric types support [arithmetic](#), [comparison](#), and [equality](#) operators.

Characteristics of the floating-point types

C# supports the following predefined floating-point types:

C# TYPE/KEYWORD	APPROXIMATE RANGE	PRECISION	SIZE	.NET TYPE
<code>float</code>	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	<code>System.Single</code>
<code>double</code>	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	<code>System.Double</code>
<code>decimal</code>	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	<code>System.Decimal</code>

In the preceding table, each C# type keyword from the leftmost column is an alias for the corresponding .NET type. They are interchangeable. For example, the following declarations declare variables of the same type:

```
double a = 12.3;  
System.Double b = 12.3;
```

The default value of each floating-point type is zero, `0`. Each of the floating-point types has the `MinValue` and `MaxValue` constants that provide the minimum and maximum finite value of that type. The `float` and `double` types also provide constants that represent not-a-number and infinity values. For example, the `double` type provides the following constants: `Double.NaN`, `Double.NegativeInfinity`, and `Double.PositiveInfinity`.

The `decimal` type is appropriate when the required degree of precision is determined by the number of digits to the right of the decimal point. Such numbers are commonly used in financial applications, for currency amounts (for example, \$1.00), interest rates (for example, 2.625%), and so forth. Even numbers that are precise to only one decimal digit are handled more accurately by the `decimal` type: 0.1, for example, can be exactly represented by a `decimal` instance, while there's no `double` or `float` instance that exactly represents 0.1. Because of this difference in numeric types, unexpected rounding errors can occur in arithmetic calculations when you use `double` or `float` for decimal data. You can use `double` instead of `decimal` when optimizing performance is more important than ensuring accuracy. However, any difference in performance would go unnoticed by all but the most calculation-intensive applications. Another possible reason to avoid `decimal` is to minimize storage requirements. For example, [ML.NET](#) uses `float` because the difference between 4 bytes and 16 bytes adds up for very large data sets. For more information, see [System.Decimal](#).

You can mix [integral](#) types and the `float` and `double` types in an expression. In this case, integral types are implicitly converted to one of the floating-point types and, if necessary, the `float` type is implicitly converted to `double`. The expression is evaluated as follows:

- If there is `double` type in the expression, the expression evaluates to `double`, or to `bool` in relational and equality comparisons.
- If there is no `double` type in the expression, the expression evaluates to `float`, or to `bool` in relational and equality comparisons.

You can also mix integral types and the `decimal` type in an expression. In this case, integral types are implicitly converted to the `decimal` type and the expression evaluates to `decimal`, or to `bool` in relational and equality comparisons.

You cannot mix the `decimal` type with the `float` and `double` types in an expression. In this case, if you want to perform arithmetic, comparison, or equality operations, you must explicitly convert the operands either from or to the `decimal` type, as the following example shows:

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

You can use either [standard numeric format strings](#) or [custom numeric format strings](#) to format a floating-point value.

Real literals

The type of a real literal is determined by its suffix as follows:

- The literal without suffix or with the `d` or `D` suffix is of type `double`
- The literal with the `f` or `F` suffix is of type `float`
- The literal with the `m` or `M` suffix is of type `decimal`

The following code demonstrates an example of each:

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

The preceding example also shows the use of `_` as a *digit separator*, which is supported starting with C# 7.0. You can use the digit separator with all kinds of numeric literals.

You can also use scientific notation, that is, specify an exponent part of a real literal, as the following example shows:

```
double d = 0.42e2;
Console.WriteLine(d); // output 42

float f = 134.45E-2f;
Console.WriteLine(f); // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m); // output: 1500000
```

Conversions

There is only one implicit conversion between floating-point numeric types: from `float` to `double`. However, you can convert any floating-point type to any other floating-point type with the [explicit cast](#). For more information, see [Built-in numeric conversions](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Floating-point types](#)
- [The decimal type](#)
- [Real literals](#)

See also

- [C# reference](#)
- [Value types](#)
- [Integral types](#)
- [Standard numeric format strings](#)
- [Numerics in .NET](#)
- [System.Numerics.Complex](#)

Built-in numeric conversions (C# reference)

3/20/2021 • 4 minutes to read • [Edit Online](#)

C# provides a set of [integral](#) and [floating-point](#) numeric types. There exists a conversion between any two numeric types, either implicit or explicit. You must use a [cast expression](#) to perform an explicit conversion.

Implicit numeric conversions

The following table shows the predefined implicit conversions between the built-in numeric types:

FROM	TO
<code>sbyte</code>	<code>short</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , <code>decimal</code> , or <code>nint</code>
<code>byte</code>	<code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , <code>decimal</code> , <code>nint</code> , or <code>nuint</code>
<code>short</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , or <code>decimal</code> , or <code>nint</code>
<code>ushort</code>	<code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , or <code>decimal</code> , <code>nint</code> , or <code>nuint</code>
<code>int</code>	<code>long</code> , <code>float</code> , <code>double</code> , or <code>decimal</code> , <code>nint</code>
<code>uint</code>	<code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , or <code>decimal</code> , or <code>nuint</code>
<code>long</code>	<code>float</code> , <code>double</code> , or <code>decimal</code>
<code>ulong</code>	<code>float</code> , <code>double</code> , or <code>decimal</code>
<code>float</code>	<code>double</code>
<code>nint</code>	<code>long</code> , <code>float</code> , <code>double</code> , or <code>decimal</code>
<code>nuint</code>	<code>ulong</code> , <code>float</code> , <code>double</code> , or <code>decimal</code>

NOTE

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

Also note that

- Any [integral numeric type](#) is implicitly convertible to any [floating-point numeric type](#).
- There are no implicit conversions to the `byte` and `sbyte` types. There are no implicit conversions from the `double` and `decimal` types.

- There are no implicit conversions between the `decimal` type and the `float` or `double` types.
- A value of a constant expression of type `int` (for example, a value represented by an integer literal) can be implicitly converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint`, or `nuint`, if it's within the range of the destination type:

```
byte a = 13;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a 'byte'
```

As the preceding example shows, if the constant value is not within the range of the destination type, a compiler error [CS0031](#) occurs.

Explicit numeric conversions

The following table shows the predefined explicit conversions between the built-in numeric types for which there is no [implicit conversion](#):

FROM	TO
<code>sbyte</code>	<code>byte</code> , <code>ushort</code> , <code>uint</code> , or <code>ulong</code> , or <code>nuint</code>
<code>byte</code>	<code>sbyte</code>
<code>short</code>	<code>sbyte</code> , <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> , or <code>nuint</code>
<code>ushort</code>	<code>sbyte</code> , <code>byte</code> , or <code>short</code>
<code>int</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> , or <code>nuint</code>
<code>uint</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , or <code>int</code>
<code>long</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>ulong</code> , <code>nint</code> , or <code>nuint</code>
<code>ulong</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>nint</code> , or <code>nuint</code>
<code>float</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>decimal</code> , <code>nint</code> , or <code>nuint</code>
<code>double</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>decimal</code> , <code>nint</code> , or <code>nuint</code>
<code>decimal</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , <code>nint</code> , or <code>nuint</code>
<code>nint</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>ulong</code> , or <code>nuint</code>
<code>nuint</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , or <code>nint</code>

NOTE

An explicit numeric conversion might result in data loss or throw an exception, typically an [OverflowException](#).

Also note that

- When you convert a value of an integral type to another integral type, the result depends on the overflow [checking context](#). In a checked context, the conversion succeeds if the source value is within the range of the destination type. Otherwise, an [OverflowException](#) is thrown. In an unchecked context, the conversion always succeeds, and proceeds as follows:
 - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
 - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it's of 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.
- When you convert a `decimal` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, an [OverflowException](#) is thrown.
- When you convert a `double` or `float` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, the result depends on the overflow [checking context](#). In a checked context, an [OverflowException](#) is thrown, while in an unchecked context, the result is an unspecified value of the destination type.
- When you convert `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small or too large to fit into the `float` type, the result is zero or infinity.
- When you convert `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. Depending on the value of the source value, one of the following results may occur:
 - If the source value is too small to be represented as a `decimal`, the result becomes zero.
 - If the source value is NaN (not a number), infinity, or too large to be represented as a `decimal`, an [OverflowException](#) is thrown.
- When you convert `decimal` to `float` or `double`, the source value is rounded to the nearest `float` or `double` value, respectively.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Implicit numeric conversions](#)
- [Explicit numeric conversions](#)

See also

- [C# reference](#)
- [Casting and type conversions](#)

bool (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `bool` type keyword is an alias for the .NET `System.Boolean` structure type that represents a Boolean value, which can be either `true` or `false`.

To perform logical operations with values of the `bool` type, use [Boolean logical operators](#). The `bool` type is the result type of [comparison](#) and [equality](#) operators. A `bool` expression can be a controlling conditional expression in the `if`, `do`, `while`, and `for` statements and in the [conditional operator](#) `?:`.

The default value of the `bool` type is `false`.

Literals

You can use the `true` and `false` literals to initialize a `bool` variable or to pass a `bool` value:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

Three-valued Boolean logic

Use the nullable `bool?` type, if you need to support the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. For the `bool?` operands, the predefined `&` and `|` operators support the three-valued logic. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For more information about nullable value types, see [Nullable value types](#).

Conversions

C# provides only two conversions that involve the `bool` type. Those are an implicit conversion to the corresponding nullable `bool?` type and an explicit conversion from the `bool?` type. However, .NET provides additional methods that you can use to convert to or from the `bool` type. For more information, see the [Converting to and from Boolean values](#) section of the [System.Boolean](#) API reference page.

C# language specification

For more information, see [The bool type](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Value types](#)
- [true and false operators](#)

char (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `char` type keyword is an alias for the .NET `System.Char` structure type that represents a Unicode UTF-16 character.

TYPE	RANGE	SIZE	.NET TYPE
<code>char</code>	U+0000 to U+FFFF	16 bit	<code>System.Char</code>

The default value of the `char` type is `\0`, that is, U+0000.

The `char` type supports [comparison](#), [equality](#), [increment](#), and [decrement](#) operators. Moreover, for `char` operands, [arithmetic](#) and [bitwise logical](#) operators perform an operation on the corresponding character codes and produce the result of the `int` type.

The `string` type represents text as a sequence of `char` values.

Literals

You can specify a `char` value with:

- a character literal.
- a Unicode escape sequence, which is `\u` followed by the four-symbol hexadecimal representation of a character code.
- a hexadecimal escape sequence, which is `\x` followed by the hexadecimal representation of a character code.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars)); // output: j j j j
```

As the preceding example shows, you can also cast the value of a character code into the corresponding `char` value.

NOTE

In the case of a Unicode escape sequence, you must specify all four hexadecimal digits. That is, `\u006A` is a valid escape sequence, while `\u06A` and `\u6A` are not valid.

In the case of a hexadecimal escape sequence, you can omit the leading zeros. That is, the `\x006A`, `\x6A`, and `\x6A` escape sequences are valid and correspond to the same character.

Conversions

The `char` type is implicitly convertible to the following [integral](#) types: `ushort`, `int`, `uint`, `long`, and `ulong`.

It's also implicitly convertible to the built-in [floating-point](#) numeric types: `float`, `double`, and `decimal`. It's explicitly convertible to `sbyte`, `byte`, and `short` integral types.

There are no implicit conversions from other types to the `char` type. However, any [integral](#) or [floating-point](#) numeric type is explicitly convertible to `char`.

C# language specification

For more information, see the [Integral types](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Value types](#)
- [Strings](#)
- [System.Text.Rune](#)
- [Character encoding in .NET](#)

Enumeration types (C# reference)

3/6/2021 • 3 minutes to read • [Edit Online](#)

An *enumeration type* (or *enum type*) is a [value type](#) defined by a set of named constants of the underlying [integral numeric](#) type. To define an enumeration type, use the `enum` keyword and specify the names of *enum members*:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

By default, the associated constant values of enum members are of type `int`; they start with zero and increase by one following the definition text order. You can explicitly specify any other [integral numeric](#) type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values, as the following example shows:

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

You cannot define a method inside the definition of an enumeration type. To add functionality to an enumeration type, create an [extension method](#).

The default value of an enumeration type `E` is the value produced by expression `(E)0`, even if zero doesn't have the corresponding enum member.

You use an enumeration type to represent a choice from a set of mutually exclusive values or a combination of choices. To represent a combination of choices, define an enumeration type as bit flags.

Enumeration types as bit flags

If you want an enumeration type to represent a combination of choices, define enum members for those choices such that an individual choice is a bit field. That is, the associated values of those enum members should be the powers of two. Then, you can use the [bitwise logical operators](#) `|` or `&` to combine choices or intersect combinations of choices, respectively. To indicate that an enumeration type declares bit fields, apply the [Flags](#) attribute to it. As the following example shows, you can also include some typical combinations in the definition of an enumeration type.

```

[Flags]
public enum Days
{
    None      = 0b_0000_0000, // 0
    Monday    = 0b_0000_0001, // 1
    Tuesday   = 0b_0000_0010, // 2
    Wednesday = 0b_0000_0100, // 4
    Thursday  = 0b_0000_1000, // 8
    Friday    = 0b_0001_0000, // 16
    Saturday  = 0b_0010_0000, // 32
    Sunday    = 0b_0100_0000, // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}

```

For more information and examples, see the [System.FlagsAttribute](#) API reference page and the [Non-exclusive members and the Flags attribute](#) section of the [System.Enum](#) API reference page.

The System.Enum type and enum constraint

The [System.Enum](#) type is the abstract base class of all enumeration types. It provides a number of methods to get information about an enumeration type and its values. For more information and examples, see the [System.Enum](#) API reference page.

Beginning with C# 7.3, you can use `System.Enum` in a base class constraint (that is known as the [enum constraint](#)) to specify that a type parameter is an enumeration type. Any enumeration type also satisfies the `struct` constraint, which is used to specify that a type parameter is a non-nullable value type.

Conversions

For any enumeration type, there exist explicit conversions between the enumeration type and its underlying integral type. If you [cast](#) an enum value to its underlying type, the result is the associated integral value of an enum member.

```

public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}"); // output: Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b); // output: Summer

        var c = (Season)4;
        Console.WriteLine(c); // output: 4
    }
}

```

Use the [Enum.IsDefined](#) method to determine whether an enumeration type contains an enum member with the certain associated value.

For any enumeration type, there exist [boxing and unboxing](#) conversions to and from the [System.Enum](#) type, respectively.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Enums](#)
- [Enum values and operations](#)
- [Enumeration logical operators](#)
- [Enumeration comparison operators](#)
- [Explicit enumeration conversions](#)
- [Implicit enumeration conversions](#)

See also

- [C# reference](#)
- [Enumeration format strings](#)
- [Design guidelines - Enum design](#)
- [Design guidelines - Enum naming conventions](#)
- [switch statement](#)

Structure types (C# reference)

3/6/2021 • 7 minutes to read • [Edit Online](#)

A *structure type* (or *struct type*) is a [value type](#) that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. In the case of a structure-type variable, an instance of the type is copied. For more information, see [Value types](#).

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both [integer](#) and [real](#)), a [Boolean value](#), a [Unicode character](#), a [time instance](#). If you're focused on the behavior of a type, consider defining a [class](#). Class types have *reference semantics*. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

Because structure types have value semantics, we recommend you to define *immutable* structure types.

readonly struct

Beginning with C# 7.2, you use the `readonly` modifier to declare that a structure type is immutable. All data members of a `readonly` struct must be read-only as follows:

- Any field declaration must have the `readonly` modifier
- Any property, including auto-implemented ones, must be read-only. In C# 9.0 and later, a property may have an `init` accessor.

That guarantees that no member of a `readonly` struct modifies the state of the struct. In C# 8.0 and later, that means that other instance members except constructors are implicitly `readonly`.

NOTE

In a `readonly` struct, a data member of a mutable reference type still can mutate its own state. For example, you can't replace a `List<T>` instance, but you can add new elements to it.

The following code defines a `readonly` struct with init-only property setters, available in C# 9.0 and later:

```

public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

```

readonly instance members

Beginning with C# 8.0, you can also use the `readonly` modifier to declare that an instance member doesn't modify the state of a struct. If you can't declare the whole structure type as `readonly`, use the `readonly` modifier to mark the instance members that don't modify the state of the struct.

Within a `readonly` instance member, you can't assign to structure's instance fields. However, a `readonly` member can call a non-`readonly` member. In that case the compiler creates a copy of the structure instance and calls the non-`readonly` member on that copy. As a result, the original structure instance is not modified.

Typically, you apply the `readonly` modifier to the following kinds of instance members:

- methods:

```

public readonly double Sum()
{
    return X + Y;
}

```

You can also apply the `readonly` modifier to methods that override methods declared in [System.Object](#):

```

public readonly override string ToString() => $"({X}, {Y})";

```

- properties and indexers:

```

private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}

```

If you need to apply the `readonly` modifier to both accessors of a property or indexer, apply it in the declaration of the property or indexer.

NOTE

The compiler declares a `get` accessor of an [auto-implemented property](#) as `readonly`, regardless of presence of the `readonly` modifier in a property declaration.

In C# 9.0 and later, you may apply the `readonly` modifier to a property or indexer with an `init`

accessor:

```
public readonly double X { get; init; }
```

You can't apply the `readonly` modifier to static members of a structure type.

The compiler may make use of the `readonly` modifier for performance optimizations. For more information, see [Write safe and efficient C# code](#).

Limitations with the design of a structure type

When you design a structure type, you have the same capabilities as with a `class` type, with the following exceptions:

- You can't declare a parameterless constructor. Every structure type already provides an implicit parameterless constructor that produces the [default value](#) of the type.
- You can't initialize an instance field or property at its declaration. However, you can initialize a `static` or `const` field or a static property at its declaration.
- A constructor of a structure type must initialize all instance fields of the type.
- A structure type can't inherit from other class or structure type and it can't be the base of a class. However, a structure type can implement [interfaces](#).
- You can't declare a [finalizer](#) within a structure type.

Instantiation of a structure type

In C#, you must initialize a declared variable before it can be used. Because a structure-type variable can't be `null` (unless it's a variable of a [nullable value type](#)), you must instantiate an instance of the corresponding type. There are several ways to do that.

Typically, you instantiate a structure type by calling an appropriate constructor with the `new` operator. Every structure type has at least one constructor. That's an implicit parameterless constructor, which produces the [default value](#) of the type. You can also use a [default value expression](#) to produce the default value of a type.

If all instance fields of a structure type are accessible, you can also instantiate it without the `new` operator. In that case you must initialize all instance fields before the first use of the instance. The following example shows how to do that:

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"{p.x}, {p.y}"); // output: (3, 4)
    }
}
```

In the case of the [built-in value types](#), use the corresponding literals to specify a value of the type.

Passing structure-type variables by reference

When you pass a structure-type variable to a method as an argument or return a structure-type value from a method, the whole instance of a structure type is copied. That can affect the performance of your code in high-performance scenarios that involve large structure types. You can avoid value copying by passing a structure-type variable by reference. Use the `ref`, `out`, or `in` method parameter modifiers to indicate that an argument must be passed by reference. Use `ref returns` to return a method result by reference. For more information, see [Write safe and efficient C# code](#).

`ref struct`

Beginning with C# 7.2, you can use the `ref` modifier in the declaration of a structure type. Instances of a `ref struct` type are allocated on the stack and can't escape to the managed heap. To ensure that, the compiler limits the usage of `ref struct` types as follows:

- A `ref struct` can't be the element type of an array.
- A `ref struct` can't be a declared type of a field of a class or a non-`ref struct`.
- A `ref struct` can't implement interfaces.
- A `ref struct` can't be boxed to [System.ValueType](#) or [System.Object](#).
- A `ref struct` can't be a type argument.
- A `ref struct` variable can't be captured by a [lambda expression](#) or a [local function](#).
- A `ref struct` variable can't be used in an `async` method. However, you can use `ref struct` variables in synchronous methods, for example, in those that return [Task](#) or [Task<TResult>](#).
- A `ref struct` variable can't be used in [iterators](#).

Typically, you define a `ref struct` type when you need a type that also includes data members of `ref struct` types:

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

To declare a `ref struct` as `readonly`, combine the `readonly` and `ref` modifiers in the type declaration (the `readonly` modifier must come before the `ref` modifier):

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

In .NET, examples of a `ref struct` are [System.Span<T>](#) and [System.ReadOnlySpan<T>](#).

struct constraint

You also use the `struct` keyword in the `struct` constraint to specify that a type parameter is a non-nullable value type. Both structure and [enumeration](#) types satisfy the `struct` constraint.

Conversions

For any structure type (except `ref struct` types), there exist [boxing and unboxing](#) conversions to and from the `System.ValueType` and `System.Object` types. There exist also boxing and unboxing conversions between a structure type and any interface that it implements.

C# language specification

For more information, see the [Structs](#) section of the [C# language specification](#).

For more information about features introduced in C# 7.2 and later, see the following feature proposal notes:

- [Readonly structs](#)
- [Readonly instance members](#)
- [Compile-time safety for ref-like types](#)

See also

- [C# reference](#)
- [Design guidelines - Choosing between class and struct](#)
- [Design guidelines - Struct design](#)
- [Classes and structs](#)

Tuple types (C# reference)

11/2/2020 • 8 minutes to read • [Edit Online](#)

Available in C# 7.0 and later, the *tuples* feature provides concise syntax to group multiple data elements in a lightweight data structure. The following example shows how you can declare a tuple variable, initialize it, and access its data members:

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

As the preceding example shows, to define a tuple type, you specify types of all its data members and, optionally, the [field names](#). You cannot define methods in a tuple type, but you can use the methods provided by .NET, as the following example shows:

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

Beginning with C# 7.3, tuple types support [equality operators](#) `==` and `!=`. For more information, see the [Tuple equality](#) section.

Tuple types are [value types](#); tuple elements are public fields. That makes tuples *mutable* value types.

NOTE

The tuples feature requires the [System.ValueTuple](#) type and related generic types (for example, [System.ValueTuple<T1,T2>](#)), which are available in .NET Core and .NET Framework 4.7 and later. To use tuples in a project that targets .NET Framework 4.6.2 or earlier, add the NuGet package [System.ValueTuple](#) to the project.

You can define tuples with an arbitrary large number of elements:

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26); // output: 26
```

Use cases of tuples

One of the most common use cases of tuples is as a method return type. That is, instead of defining [out](#) [method parameters](#), you can group method results in a tuple return type, as the following example shows:

```

var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and {limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and {maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empty array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}

```

As the preceding example shows, you can work with the returned tuple instance directly or [deconstruct](#) it in separate variables.

You can also use tuple types instead of [anonymous types](#); for example, in LINQ queries. For more information, see [Choosing between anonymous and tuple types](#).

Typically, you use tuples to group loosely related data elements. That is usually useful within private and internal utility methods. In the case of public API, consider defining a [class](#) or a [structure](#) type.

Tuple field names

You can explicitly specify the names of tuple fields either in a tuple initialization expression or in the definition of a tuple type, as the following example shows:

```

var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");

```

Beginning with C# 7.1, if you don't specify a field name, it may be inferred from the name of the corresponding variable in a tuple initialization expression, as the following example shows:

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

That's known as tuple projection initializers. The name of a variable isn't projected onto a tuple field name in the following cases:

- The candidate name is a member name of a tuple type, for example, `Item3`, `ToString`, or `Rest`.
- The candidate name is a duplicate of another tuple field name, either explicit or implicit.

In those cases you either explicitly specify the name of a field or access a field by its default name.

The default names of tuple fields are `Item1`, `Item2`, `Item3` and so on. You can always use the default name of a field, even when a field name is specified explicitly or inferred, as the following example shows:

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

[Tuple assignment](#) and [tuple equality comparisons](#) don't take field names into account.

At compile time, the compiler replaces non-default field names with the corresponding default names. As a result, explicitly specified or inferred field names aren't available at run time.

Tuple assignment and deconstruction

C# supports assignment between tuple types that satisfy both of the following conditions:

- both tuple types have the same number of elements
- for each tuple position, the type of the right-hand tuple element is the same as or implicitly convertible to the type of the corresponding left-hand tuple element

Tuple element values are assigned following the order of tuple elements. The names of tuple fields are ignored and not assigned, as the following example shows:

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

You can also use the assignment operator `=` to *deconstruct* a tuple instance in separate variables. You can do that in one of the following ways:

- Explicitly declare the type of each variable inside parentheses:

```
var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Use the `var` keyword outside the parentheses to declare implicitly typed variables and let the compiler infer their types:

```
var t = ("post office", 3.6);
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Use existing variables:

```
var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

For more information about deconstruction of tuples and other types, see [Deconstructing tuples and other types](#).

Tuple equality

Beginning with C# 7.3, tuple types support the `==` and `!=` operators. These operators compare members of the left-hand operand with the corresponding members of the right-hand operand following the order of tuple elements.

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right); // output: True
Console.WriteLine(left != right); // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2); // output: True
Console.WriteLine(t1 != t2); // output: False
```

As the preceding example shows, the `==` and `!=` operations don't take into account tuple field names.

Two tuples are comparable when both of the following conditions are satisfied:

- Both tuples have the same number of elements. For example, `t1 != t2` doesn't compile if `t1` and `t2` have different numbers of elements.
- For each tuple position, the corresponding elements from the left-hand and right-hand tuple operands are comparable with the `==` and `!=` operators. For example, `(1, (2, 3)) == ((1, 2), 3)` doesn't compile because `1` is not comparable with `(1, 2)`.

The `==` and `!=` operators compare tuples in short-circuiting way. That is, an operation stops as soon as it meets a pair of non equal elements or reaches the ends of tuples. However, before any comparison, *all* tuple elements are evaluated, as the following example shows:

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}
// Output:
// 1
// 2
// 3
// 4
// False
```

Tuples as out parameters

Typically, you refactor a method that has `out` parameters into a method that returns a tuple. However, there are cases in which an `out` parameter can be of a tuple type. The following example shows how to work with tuples as `out` parameters:

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is {limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

Tuples vs `System.Tuple`

C# tuples, which are backed by `System.ValueTuple` types, are different from tuples that are represented by `System.Tuple` types. The main differences are as follows:

- `ValueTuple` types are **value types**. `Tuple` types are **reference types**.
- `ValueTuple` types are mutable. `Tuple` types are immutable.
- Data members of `ValueTuple` types are fields. Data members of `Tuple` types are properties.

C# language specification

For more information, see the following feature proposal notes:

- [Infer tuple names \(aka. tuple projection initializers\)](#)
- [Support for `==` and `!=` on tuple types](#)

See also

- [C# reference](#)

- [Value types](#)
- [Choosing between anonymous and tuple types](#)
- [System.ValueTuple](#)

Nullable value types (C# reference)

11/2/2020 • 7 minutes to read • [Edit Online](#)

A *nullable value type* `T?` represents all values of its underlying *value type* `T` and an additional `null` value. For example, you can assign any of the following three values to a `bool?` variable: `true`, `false`, or `null`. An underlying value type `T` cannot be a nullable value type itself.

NOTE

C# 8.0 introduces the nullable reference types feature. For more information, see [Nullable reference types](#). The nullable value types are available beginning with C# 2.

Any nullable value type is an instance of the generic `System.Nullable<T>` structure. You can refer to a nullable value type with an underlying type `T` in any of the following interchangeable forms: `Nullable<T>` or `T?`.

You typically use a nullable value type when you need to represent the undefined value of an underlying value type. For example, a Boolean, or `bool`, variable can only be either `true` or `false`. However, in some applications a variable value can be undefined or missing. For example, a database field may contain `true` or `false`, or it may contain no value at all, that is, `NULL`. You can use the `bool?` type in that scenario.

Declaration and assignment

As a value type is implicitly convertible to the corresponding nullable value type, you can assign a value to a variable of a nullable value type as you would do that for its underlying value type. You can also assign the `null` value. For example:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int[][] arr = new int?[10];
```

The default value of a nullable value type represents `null`, that is, it's an instance whose `Nullable<T>.HasValue` property returns `false`.

Examination of an instance of a nullable value type

Beginning with C# 7.0, you can use the `is` operator with a type pattern to both examine an instance of a nullable value type for `null` and retrieve a value of an underlying type:

```

int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42

```

You always can use the following read-only properties to examine and get a value of a nullable value type variable:

- `Nullable<T>.HasValue` indicates whether an instance of a nullable value type has a value of its underlying type.
- `Nullable<T>.Value` gets the value of an underlying type if `HasValue` is `true`. If `HasValue` is `false`, the `Value` property throws an `InvalidOperationException`.

The following example uses the `.HasValue` property to test whether the variable contains a value before displaying it:

```

int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10

```

You can also compare a variable of a nullable value type with `null` instead of using the `.HasValue` property, as the following example shows:

```

int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7

```

Conversion from a nullable value type to an underlying type

If you want to assign a value of a nullable value type to a non-nullable value type variable, you might need to specify the value to be assigned in place of `null`. Use the `null-coalescing operator ??` to do that (you can also use the `Nullable<T>.GetValueOrDefault(T)` method for the same purpose):

```

int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}"); // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}"); // output: d is -1

```

If you want to use the [default](#) value of the underlying value type in place of `null`, use the `Nullable<T>.GetValueOrDefault()` method.

You can also explicitly cast a nullable value type to a non-nullable type, as the following example shows:

```

int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null

```

At run time, if the value of a nullable value type is `null`, the explicit cast throws an [InvalidOperationException](#).

A non-nullable value type `T` is implicitly convertible to the corresponding nullable value type `T?`.

Lifted operators

The predefined unary and binary [operators](#) or any overloaded operators that are supported by a value type `T` are also supported by the corresponding nullable value type `T?`. These operators, also known as *lifted operators*, produce `null` if one or both operands are `null`; otherwise, the operator uses the contained values of its operands to calculate the result. For example:

```

int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c; // a is 110
a = a + b; // a is null

```

NOTE

For the `bool?` type, the predefined `&` and `|` operators don't follow the rules described in this section: the result of an operator evaluation can be non-null even if one of the operands is `null`. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For the [comparison operators](#) `<`, `>`, `<=`, and `>=`, if one or both operands are `null`, the result is `false`; otherwise, the contained values of operands are compared. Do not assume that because a particular comparison (for example, `<=`) returns `false`, the opposite comparison (`>`) returns `true`. The following example shows that 10 is

- neither greater than or equal to `null`
- nor less than `null`

```

int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True

```

For the [equality operator](#) `==`, if both operands are `null`, the result is `true`, if only one of the operands is `null`, the result is `false`; otherwise, the contained values of operands are compared.

For the [inequality operator](#) `!=`, if both operands are `null`, the result is `false`, if only one of the operands is `null`, the result is `true`; otherwise, the contained values of operands are compared.

If there exists a [user-defined conversion](#) between two value types, the same conversion can also be used between the corresponding nullable value types.

Boxing and unboxing

An instance of a nullable value type `T?` is [boxed](#) as follows:

- If `HasValue` returns `false`, the null reference is produced.
- If `HasValue` returns `true`, the corresponding value of the underlying value type `T` is boxed, not the instance of `Nullable<T>`.

You can unbox a boxed value of a value type `T` to the corresponding nullable value type `T?`, as the following example shows:

```

int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41

```

How to identify a nullable value type

The following example shows how to determine whether a `System.Type` instance represents a constructed nullable value type, that is, the `System.Nullable<T>` type with a specified type parameter `T`:

```

Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non nullable value type

```

As the example shows, you use the `typeof` operator to create a `System.Type` instance.

If you want to determine whether an instance is of a nullable value type, don't use the `Object.GetType` method to get a `Type` instance to be tested with the preceding code. When you call the `Object.GetType` method on an instance of a nullable value type, the instance is [boxed](#) to `Object`. As boxing of a non-null instance of a nullable value type is equivalent to boxing of a value of the underlying type, `GetType` returns a `Type` instance that represents the underlying type of a nullable value type:

```

int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32

```

Also, don't use the `is` operator to determine whether an instance is of a nullable value type. As the following example shows, you cannot distinguish types of a nullable value type instance and its underlying type instance with the `is` operator:

```

int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?

```

You can use the code presented in the following example to determine whether an instance is of a nullable value type:

```

int? a = 14;
Console.WriteLine(IsOfNullableType(a)); // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b)); // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}

```

NOTE

The methods described in this section are not applicable in the case of [nullable reference types](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Nullable types](#)
- [Lifted operators](#)
- [Implicit nullable conversions](#)
- [Explicit nullable conversions](#)
- [Lifted conversion operators](#)

See also

- [C# reference](#)
- [What exactly does 'lifted' mean?](#)
- [System.Nullable<T>](#)
- [System.Nullable](#)
- [Nullable.GetUnderlyingType](#)
- [Nullable reference types](#)

Reference types (C# Reference)

4/6/2021 • 2 minutes to read • [Edit Online](#)

There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of `in`, `ref` and `out` parameter variables; see [in](#), [ref](#) and [out](#) parameter modifier).

The following keywords are used to declare reference types:

- [class](#)
- [interface](#)
- [delegate](#)
- [record](#)

C# also provides the following built-in reference types:

- [dynamic](#)
- [object](#)
- [string](#)

See also

- [C# Reference](#)
- [C# Keywords](#)
- [Pointer types](#)
- [Value types](#)

Built-in reference types (C# reference)

3/12/2020 • 6 minutes to read • [Edit Online](#)

C# has a number of built-in reference types. They have keywords or operators that are synonyms for a type in the .NET library.

The object type

The `object` type is an alias for `System.Object` in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from `System.Object`. You can assign values of any type to variables of type `object`. Any `object` variable can be assigned to its default value using the literal `null`. When a variable of a value type is converted to `object`, it is said to be *boxed*. When a variable of type `object` is converted to a value type, it is said to be *unboxed*. For more information, see [Boxing and Unboxing](#).

The string type

The `string` type represents a sequence of zero or more Unicode characters. `string` is an alias for `System.String` in .NET.

Although `string` is a reference type, the [equality operators](#) `==` and `!=` are defined to compare the values of `string` objects, not references. This makes testing for string equality more intuitive. For example:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

This displays "True" and then "False" because the content of the strings are equivalent, but `a` and `b` do not refer to the same string instance.

The [+ operator](#) concatenates strings:

```
string a = "good " + "morning";
```

This creates a string object that contains "good morning".

Strings are *immutable*--the contents of a string object cannot be changed after the object is created, although the syntax makes it appear as if you can do this. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to `b`. The memory that had been allocated for `b` (when it contained the string "h") is then eligible for garbage collection.

```
string b = "h";
b += "ello";
```

The [\[\] operator](#) can be used for readonly access to individual characters of a string. Valid index values start at `0` and must be less than the length of the string:

```
string str = "test";
char x = str[2]; // x = 's';
```

In similar fashion, the `[]` operator can also be used for iterating over each character in a string:

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
    Console.Write(str[i] + " ");
}
// Output: t e s t
```

String literals are of type `string` and can be written in two forms, quoted and `@`-quoted. Quoted string literals are enclosed in double quotation marks (""):

```
"good morning" // a string literal
```

String literals can contain any character literal. Escape sequences are included. The following example uses escape sequence `\\"` for backslash, `\u0066` for the letter f, and `\n` for newline.

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
// F
```

NOTE

The escape code `\udddd` (where `ddd` is a four-digit number) represents the Unicode character U+`ddd`. Eight-digit Unicode escape codes are also recognized: `\Udddddd`.

[Verbatim string literals](#) start with `@` and are also enclosed in double quotation marks. For example:

```
@"good morning" // a string literal
```

The advantage of verbatim strings is that escape sequences are *not* processed, which makes it easy to write, for example, a fully qualified Windows file name:

```
@"c:\Docs\Source\a.txt" // rather than "c:\\Docs\\\\Source\\\\a.txt"
```

To include a double quotation mark in an `@`-quoted string, double it:

```
"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

The delegate type

The declaration of a delegate type is similar to a method signature. It has a return value and any number of parameters of any type:

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

In .NET, `System.Action` and `System.Func` types provide generic definitions for many common delegates. You likely don't need to define new custom delegate types. Instead, you can create instantiations of the provided generic types.

A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see [Delegates](#) and [Generic Delegates](#). Delegates are the basis for [Events](#). A delegate can be instantiated by associating it either with a named or anonymous method.

The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters. For more information on the degree of variance that is allowed in the method signature, see [Variance in Delegates](#). For use with anonymous methods, the delegate and the code to be associated with it are declared together.

The dynamic type

The `dynamic` type indicates that use of the variable and references to its members bypass compile-time type checking. Instead, these operations are resolved at run time. The `dynamic` type simplifies access to COM APIs such as the Office Automation APIs, to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type `dynamic` behaves like type `object` in most circumstances. In particular, any non-null expression can be converted to the `dynamic` type. The `dynamic` type differs from `object` in that operations that contain expressions of type `dynamic` are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type `dynamic` are compiled into variables of type `object`. Therefore, type `dynamic` exists only at compile time, not at run time.

The following example contrasts a variable of type `dynamic` to a variable of type `object`. To verify the type of each variable at compile time, place the mouse pointer over `dyn` or `obj` in the `WriteLine` statements. Copy the following code into an editor where IntelliSense is available. IntelliSense shows `dynamic` for `dyn` and `object` for `obj`.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

The `WriteLine` statements display the run-time types of `dyn` and `obj`. At that point, both have the same type, integer. The following output is produced:

```
System.Int32  
System.Int32
```

To see the difference between `dyn` and `obj` at compile time, add the following two lines between the declarations and the `WriteLine` statements in the previous example.

```
dyn = dyn + 3;  
obj = obj + 3;
```

A compiler error is reported for the attempted addition of an integer and an object in expression `obj + 3`. However, no error is reported for `dyn + 3`. The expression that contains `dyn` is not checked at compile time because the type of `dyn` is `dynamic`.

The following example uses `dynamic` in several declarations. The `Main` method also contrasts compile-time type checking with run-time type checking.

```

using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}

// Results:
// Local variable
// 2
// Local variable

```

See also

- [C# Reference](#)
- [C# Keywords](#)
- [Events](#)
- [Using Type dynamic](#)
- [Best Practices for Using Strings](#)
- [Basic String Operations](#)
- [Creating New Strings](#)
- [Type-testing and cast operators](#)
- [How to safely cast using pattern matching and the as and is operators](#)

- Walkthrough: creating and using dynamic objects
- [System.Object](#)
- [System.String](#)
- [System.Dynamic.DynamicObject](#)

Records (C# reference)

3/23/2021 • 14 minutes to read • [Edit Online](#)

Beginning with C# 9, you use the `record` keyword to define a [reference type](#) that provides built-in functionality for encapsulating data. You can create record types with immutable properties by using positional parameters or standard property syntax:

```
public record Person(string FirstName, string LastName);
```

```
public record Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
};
```

You can also create record types with mutable properties and fields:

```
public record Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
};
```

While records can be mutable, they are primarily intended for supporting immutable data models. The record type offers the following features:

- [Concise syntax for creating a reference type with immutable properties](#)
- Built-in behavior useful for a data-centric reference type:
 - [Value equality](#)
 - [Concise syntax for nondestructive mutation](#)
 - [Built-in formatting for display](#)
- [Support for inheritance hierarchies](#)

You can also use [structure types](#) to design data-centric types that provide value equality and little or no behavior. But for relatively large data models, structure types have some disadvantages:

- They don't support inheritance.
- They're less efficient at determining value equality. For value types, the `ValueType.Equals` method uses reflection to find all fields. For records, the compiler generates the `Equals` method. In practice, the implementation of value equality in records is measurably faster.
- They use more memory in some scenarios, since every instance has a complete copy of all of the data. Record types are [reference types](#), so a record instance contains only a reference to the data.

Positional syntax for property definition

You can use positional parameters to declare properties of a record and to initialize the property values when you create an instance:

```

public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}

```

When you use the positional syntax for property definition, the compiler creates:

- A public init-only auto-implemented property for each positional parameter provided in the record declaration. An [init-only](#) property can only be set in the constructor or by using a property initializer.
- A primary constructor whose parameters match the positional parameters on the record declaration.
- A `Deconstruct` method with an `out` parameter for each positional parameter provided in the record declaration. This method is provided only if there are two or more positional parameters. The method deconstructs properties defined by using positional syntax; it ignores properties that are defined by using standard property syntax.

If the generated auto-implemented property definition isn't what you want, you can define your own property of the same name. If you do that, the generated constructor and deconstructor will use your property definition. For instance, the following example makes the `FirstName` positional property `internal` instead of `public`.

```

public record Person(string FirstName, string LastName)
{
    internal string FirstName { get; init; } = FirstName;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person.FirstName); //output: Nancy
}

```

A record type doesn't have to declare any positional properties. You can declare a record without any positional properties, and you can declare additional fields and properties, as in the following example:

```

public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
};

```

If you define properties by using standard property syntax but omit the access modifier, the properties are implicitly `public`.

Immutability

A record type is not necessarily immutable. You can declare properties with `set` accessors and fields that aren't `readonly`. But while records can be mutable, they make it easier to create immutable data models.

Immutability can be useful when you need a data-centric type to be thread-safe or you're depending on a hash code remaining the same in a hash table. Immutability isn't appropriate for all data scenarios, however. [Entity Framework Core](#), for example, doesn't support updating with immutable entity types.

Init-only properties, whether created from positional parameters or by specifying `init` accessors, have *shallow immutability*. After initialization, you can't change the value of value-type properties or the reference of

reference-type properties. However, the data that a reference-type property refers to can be changed. The following example shows that the content of a reference-type immutable property (an array in this case) is mutable:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state.

Value equality

Value equality means that two variables of a record type are equal if the types match and all property and field values match. For other reference types, equality means identity. That is, two variables of a reference type are equal if they refer to the same object.

Reference equality is required for some data models. For example, [Entity Framework Core](#) depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, record types aren't appropriate for use as entity types in Entity Framework Core.

The following example illustrates value equality of record types:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

To implement value equality, the compiler synthesizes the following methods:

- An override of [Object.Equals\(Object\)](#).

This method is used as the basis for the [Object.Equals\(Object, Object\)](#) static method when both parameters are non-null.

- A virtual `Equals` method whose parameter is the record type. This method implements [IEquatable<T>](#).
- An override of [Object.GetHashCode\(\)](#).
- Overrides of operators `==` and `!=`.

In `class` types, you could manually override equality methods and operators to achieve value equality, but developing and testing that code would be time-consuming and error-prone. Having this functionality built-in

prevents bugs that would result from forgetting to update custom override code when properties or fields are added or changed.

You can write your own implementations to replace any of these synthesized methods. If a record type has a method that matches the signature of any synthesized method, the compiler doesn't synthesize that method.

If you provide your own implementation of `Equals` in a record type, provide an implementation of `GetHashCode` also.

Nondestructive mutation

If you need to mutate immutable properties of a record instance, you can use a `with` expression to achieve *nondestructive mutation*. A `with` expression makes a new record instance that is a copy of an existing record instance, with specified properties and fields modified. You use [object initializer](#) syntax to specify the values to be changed, as shown in the following example:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

The `with` expression can set positional properties or properties created by using standard property syntax. Non-positional properties must have an `init` or `set` accessor to be changed in a `with` expression.

The result of a `with` expression is a *shallow copy*, which means that for a reference property, only the reference to an instance is copied. Both the original record and the copy end up with a reference to the same instance.

To implement this feature, the compiler synthesizes a clone method and a copy constructor. The constructor takes an instance of the record to be copied and calls the clone method. When you use a `with` expression, the compiler creates code that calls the copy constructor and then sets the properties that are specified in the `with` expression.

If you need different copying behavior, you can write your own copy constructor. If you do that, the compiler won't synthesize one. Make your constructor `private` if the record is `sealed`, otherwise make it `protected`.

You can't override the clone method, and you can't create a member named `Clone`. The actual name of the clone method is compiler-generated.

Built-in formatting for display

Record types have a compiler-generated [ToString](#) method that displays the names and values of public properties and fields. The `ToString` method returns a string of the following format:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

For reference types, the type name of the object that the property refers to is displayed instead of the property value. In the following example, the array is a reference type, so `System.String[]` is displayed instead of the actual array element values:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

To implement this feature, the compiler synthesizes a virtual `PrintMembers` method and a `ToString` override. The `ToString` override creates a `StringBuilder` object with the type name followed by an opening bracket. It calls `PrintMembers` to add property names and values, then adds the closing bracket. The following example shows code similar to what the synthesized override contains:

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

You can provide your own implementation of `PrintMembers` or the `ToString` override. Examples are provided in the [PrintMembers formatting in derived records](#) section later in this article.

Inheritance

A record can inherit from another record. However, a record can't inherit from a class, and a class can't inherit from a record.

Positional parameters in derived record types

The derived record declares positional parameters for all the parameters in the base record primary constructor. The base record declares and initializes those properties. The derived record doesn't hide them, but only creates and initializes properties for parameters that aren't declared in its base record.

The following example illustrates inheritance with positional property syntax:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Equality in inheritance hierarchies

For two record variables to be equal, the run-time type must be equal. The types of the containing variables

might be different. This is illustrated in the following code example:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

In the example, all instances have the same properties and the same property values. But `student == teacher` returns `False` although both are `Person`-type variables, and `student == student2` returns `True` although one is a `Person` variable and one is a `Student` variable.

To implement this behavior, the compiler synthesizes an `EqualityContract` property that returns a `Type` object that matches the type of the record. This enables the equality methods to compare the runtime type of objects when they are checking for equality. If the base type of a record is `object`, this property is `virtual`. If the base type is another record type, this property is an override. If the record type is `sealed`, this property is `sealed`.

When comparing two instances of a derived type, the synthesized equality methods check all properties of the base and derived types for equality. The synthesized `GetHashCode` method uses the `GetHashCode` method from all properties and fields declared in the base type and the derived record type.

with expressions in derived records

Because the synthesized clone method uses a [covariant return type](#), the result of a `with` expression has the same run-time type as the expression's operand. All properties of the run-time type get copied, but you can only set properties of the compile-time type, as the following example shows:

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};

public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or Zderived
    Console.WriteLine(p2 is NamedPoint); // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4 }

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7, Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8 }
}
```

The synthesized `PrintMembers` method of a derived record type calls the base implementation. The result is that all public properties and fields of both derived and base types are included in the `ToString` output, as shown in the following example:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

You can provide your own implementation of the `PrintMembers` method. If you do that, use the following signature:

- For a `sealed` record that derives from `object` (doesn't declare a base record):

```
private bool PrintMembers(StringBuilder builder);
```

- For a `sealed` record that derives from another record:

```
protected sealed override bool PrintMembers(StringBuilder builder);
```

- For a record that isn't `sealed` and derives from `object`:

```
protected virtual bool PrintMembers(StringBuilder builder);
```

- For a record that isn't `sealed` and derives from another record:

```
protected override bool PrintMembers(StringBuilder builder);
```

Here is an example of code that replaces the synthesized `PrintMembers` methods, one for a record type that derives from `object`, and one for a record type that derives from another record:

```

public abstract record Person(string FirstName, string LastName, string[] PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName = {LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]}, PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[] PhoneNumbers, int Grade)
: Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1 = 555-1234, PhoneNumber2 = 555-6789, Grade = 3 }
}

```

Deconstructor behavior in derived records

The `Deconstruct` method of a derived record returns the values of all positional properties of the compile-time type. If the variable type is a base record, only the base record properties are deconstructed unless the object is cast to the derived type. The following example demonstrates calling a deconstructor on a derived record.

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
: Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
: Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy, Davolio, 3
}

```

Generic constraints

There is no generic constraint that requires a type to be a record. Records satisfy the `class` constraint. To make a constraint on a specific hierarchy of record types, put the constraint on the base record as you would a base class. For more information, see [Constraints on type parameters](#).

C# language specification

For more information, see the [Classes](#) section of the [C# language specification](#).

For more information about features introduced in C# 9 and later, see the following feature proposal notes:

- [Records](#)
- [Init-only setters](#)
- [Covariant returns](#)

See also

- [C# reference](#)
- [Design guidelines - Choosing between class and struct](#)
- [Design guidelines - Struct design](#)
- [Classes and structs](#)
- [with expression](#)

class (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Classes are declared using the keyword `class`, as shown in the following example:

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Remarks

Only single inheritance is allowed in C#. In other words, a class can inherit implementation from one base class only. However, a class can implement more than one interface. The following table shows examples of class inheritance and interface implementation:

INHERITANCE	EXAMPLE
None	<code>class ClassA { }</code>
Single	<code>class DerivedClass : BaseClass { }</code>
None, implements two interfaces	<code>class ImplClass : IFace1, IFace2 { }</code>
Single, implements one interface	<code>class ImplDerivedClass : BaseClass, IFace1 { }</code>

Classes that you declare directly within a namespace, not nested within other classes, can be either [public](#) or [internal](#). Classes are [internal](#) by default.

Class members, including nested classes, can be [public](#), [protected internal](#), [protected](#), [internal](#), [private](#), or [private protected](#). Members are [private](#) by default.

For more information, see [Access Modifiers](#).

You can declare generic classes that have type parameters. For more information, see [Generic Classes](#).

A class can contain declarations of the following members:

- [Constructors](#)
- [Constants](#)
- [Fields](#)
- [Finalizers](#)
- [Methods](#)
- [Properties](#)
- [Indexers](#)
- [Operators](#)

- [Events](#)
- [Delegates](#)
- [Classes](#)
- [Interfaces](#)
- [Structure types](#)
- [Enumeration types](#)

Example

The following example demonstrates declaring class fields, constructors, and methods. It also demonstrates object instantiation and printing instance data. In this example, two classes are declared. The first class, `Child`, contains two private fields (`name` and `age`), two public constructors and one public method. The second class, `StringTest`, is used to contain `Main`.

```

class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
   Child #1: Craig, 11 years old.
   Child #2: Sally, 10 years old.
   Child #3: N/A, 0 years old.
*/

```

Comments

Notice that in the previous example the private fields (`name` and `age`) can only be accessed through the public method of the `Child` class. For example, you cannot print the child's name, from the `Main` method, using a statement like this:

```
Console.WriteLine(child1.name); // Error
```

Accessing private members of `Child` from `Main` would only be possible if `Main` were a member of the class.

Types declared inside a class without an access modifier default to `private`, so the data members in this

example would still be `private` if the keyword were removed.

Finally, notice that for the object created using the parameterless constructor (`child3`), the `age` field was initialized to zero by default.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Reference Types](#)

interface (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

An interface defines a contract. Any [class](#) or [struct](#) that implements that contract must provide an implementation of the members defined in the interface. Beginning with C# 8.0, an interface may define a default implementation for members. It may also define [static](#) members in order to provide a single implementation for common functionality.

In the following example, class `ImplementationClass` must implement a method named `SampleMethod` that has no parameters and returns `void`.

For more information and examples, see [Interfaces](#).

Example

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

An interface can be a member of a namespace or a class. An interface declaration can contain declarations (signatures without any implementation) of the following members:

- [Methods](#)
- [Properties](#)
- [Indexers](#)
- [Events](#)

These preceding member declarations typically do not contain a body. Beginning with C# 8.0, an interface member may declare a body. This is called a *default implementation*. Members with bodies permit the interface to provide a "default" implementation for classes and structs that don't provide an overriding implementation. In addition, beginning with C# 8.0, an interface may include:

- [Constants](#)
- [Operators](#)
- [Static constructor](#).

- [Nested types](#)
- [Static fields, methods, properties, indexers, and events](#)
- Member declarations using the explicit interface implementation syntax.
- Explicit access modifiers (the default access is `public`).

Interfaces may not contain instance state. While static fields are now permitted, instance fields are not permitted in interfaces. [Instance auto-properties](#) are not supported in interfaces, as they would implicitly declare a hidden field. This rule has a subtle effect on property declarations. In an interface declaration, the following code does not declare an auto-implemented property as it does in a `class` or `struct`. Instead, it declares a property that doesn't have a default implementation but must be implemented in any type that implements the interface:

```
public interface INamed
{
    public string Name {get; set;}
}
```

An interface can inherit from one or more base interfaces. When an interface [overrides a method](#) implemented in a base interface, it must use the [explicit interface implementation](#) syntax.

When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface. In addition, default interface members can only be accessed through an instance of the interface.

For more information about explicit interface implementation, see [Explicit Interface Implementation](#).

Example

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements `IPoint` has integer properties `x` and `y`.

```

interface IPoint
{
    // Property signatures:
    int X
    {
        get;
        set;
    }

    int Y
    {
        get;
        set;
    }

    double Distance
    {
        get;
    }
}

class Point : IPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
        Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3

```

C# language specification

For more information, see the [Interfaces](#) section of the [C# language specification](#) and the feature specification for [Default interface members - C# 8.0](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Reference Types](#)
- [Interfaces](#)
- [Using Properties](#)
- [Using Indexers](#)

Nullable reference types (C# reference)

3/6/2021 • 5 minutes to read • [Edit Online](#)

NOTE

This article covers nullable reference types. You can also declare [nullable value types](#).

Nullable reference types are available beginning with C# 8.0, in code that has opted in to a *nullable aware context*. Nullable reference types, the null static analysis warnings, and the [null-forgiving operator](#) are optional language features. All are turned off by default. A *nullable context* is controlled at the project level using build settings, or in code using pragmas.

In a nullable aware context:

- A variable of a reference type `T` must be initialized with non-null, and may never be assigned a value that may be `null`.
- A variable of a reference type `T?` may be initialized with `null` or assigned `null`, but is required to be checked against `null` before de-referencing.
- A variable `m` of type `T?` is considered to be non-null when you apply the null-forgiving operator, as in `m!`.

The distinctions between a non-nullable reference type `T` and a nullable reference type `T?` are enforced by the compiler's interpretation of the preceding rules. A variable of type `T` and a variable of type `T?` are represented by the same .NET type. The following example declares a non-nullable string and a nullable string, and then uses the null-forgiving operator to assign a value to a non-nullable string:

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

The variables `notNull` and `nullable` are both represented by the [String](#) type. Because the non-nullable and nullable types are both stored as the same type, there are several locations where using a nullable reference type isn't allowed. In general, a nullable reference type can't be used as a base class or implemented interface. A nullable reference type can't be used in any object creation or type testing expression. A nullable reference type can't be the type of a member access expression. The following examples show these constructs:

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

Nullable references and static analysis

The examples in the previous section illustrate the nature of nullable reference types. Nullable reference types aren't new class types, but rather annotations on existing reference types. The compiler uses those annotations to help you find potential null reference errors in your code. There's no runtime difference between a non-nullable reference type and a nullable reference type. The compiler doesn't add any runtime checking for non-nullable reference types. The benefits are in the compile-time analysis. The compiler generates warnings that help you find and fix potential null errors in your code. You declare your intent, and the compiler warns you when your code violates that intent.

In a nullable enabled context, the compiler performs static analysis on variables of any reference type, both nullable and non-nullable. The compiler tracks the null state of each reference variable as either *not null* or *maybe null*. The default state of a non-nullable reference is *not null*. The default state of a nullable reference is *maybe null*.

Non-nullable reference types should always be safe to dereference because their null state is *not null*. To enforce that rule, the compiler issues warnings if a non-nullable reference type isn't initialized to a non-null value. Local variables must be assigned where they're declared. Every constructor must assign every field, either in its body, a called constructor, or using a field initializer. The compiler issues warnings if a non-nullable reference is assigned to a reference whose state is *maybe null*. However, because a non-nullable reference is *not null*, no warnings are issued when those variables are de-referenced.

Nullable reference types may be initialized or assigned to `null`. Therefore, static analysis must determine that a variable is *not null* before it's dereferenced. If a nullable reference is determined to be *maybe null*, it can't be assigned to a non-nullable reference variable. The following class shows examples of these warnings:

```

public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! short description not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}

```

The following snippet shows where the compiler emits warnings when using this class:

```

string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis knows shortDescription
maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");

```

The preceding examples demonstrate the compiler's static analysis to determine the null state of reference variables. The compiler applies language rules for null checks and assignments to inform its analysis. The compiler can't make assumptions about the semantics of methods or properties. If you call methods that perform null checks, the compiler can't know those methods affect a variable's null state. There are a number of attributes you can add to your APIs to inform the compiler about the semantics of arguments and return values. These attributes have been applied to many common APIs in the .NET Core libraries. For example, [IsNullOrEmpty](#) has been updated, and the compiler correctly interprets that method as a null check. For more

information about the attributes that apply to null state static analysis, see the article on [Nullable attributes](#).

Setting the nullable context

There are two ways to control the nullable context. At the project level, you can add the `<Nullable>enable</Nullable>` project setting. In a single C# source file, you can add the `#nullable enable` pragma to enable the nullable context. See the article on [setting a nullable strategy](#).

C# language specification

For more information, see the following proposals for the [C# language specification](#):

- [Nullable reference types](#)
- [Draft nullable reference types specification](#)

See also

- [C# reference](#)
- [Nullable value types](#)

void (C# reference)

4/6/2021 • 2 minutes to read • [Edit Online](#)

You use `void` as the return type of a [method](#) (or a [local function](#)) to specify that the method doesn't return a value.

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

You can also use `void` as a referent type to declare a pointer to an unknown type. For more information, see [Pointer types](#).

You cannot use `void` as the type of a variable.

See also

- [C# reference](#)
- [System.Void](#)

var (C# reference)

4/7/2021 • 2 minutes to read • [Edit Online](#)

Beginning with C# 3, variables that are declared at method scope can have an implicit "type" `var`. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type. The following two declarations of `i` are functionally equivalent:

```
var i = 10; // Implicitly typed.  
int i = 10; // Explicitly typed.
```

IMPORTANT

When `var` is used with [nullable reference types](#) enabled, it always implies a nullable reference type even if the expression type isn't nullable.

A common use of the `var` keyword is with constructor invocation expressions. The use of `var` allows you to not repeat a type name in a variable declaration and object instantiation, as the following example shows:

```
var xs = new List<int>();
```

Beginning with C# 9.0, you can use a target-typed `new` expression as an alternative:

```
List<int> xs = new();  
List<int>? ys = new();
```

In pattern matching, the `var` keyword is used in a `var` pattern.

Example

The following example shows two query expressions. In the first expression, the use of `var` is permitted but is not required, because the type of the query result can be stated explicitly as an `IEnumerable<string>`. However, in the second expression, `var` allows the result to be a collection of anonymous types, and the name of that type is not accessible except to the compiler itself. Use of `var` eliminates the requirement to create a new class for the result. Note that in Example #2, the `foreach` iteration variable `item` must also be implicitly typed.

```
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
    where word[0] == 'g'
    select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
    where cust.City == "Phoenix"
    select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

See also

- [C# reference](#)
- [Implicitly typed local variables](#)
- [Type relationships in LINQ query operations](#)

Built-in types (C# reference)

3/27/2021 • 2 minutes to read • [Edit Online](#)

The following table lists the C# built-in [value](#) types:

C# TYPE KEYWORD	.NET TYPE
<code>bool</code>	<code>System.Boolean</code>
<code>byte</code>	<code>System.Byte</code>
<code>sbyte</code>	<code>System.SByte</code>
<code>char</code>	<code>System.Char</code>
<code>decimal</code>	<code>System.Decimal</code>
<code>double</code>	<code>System.Double</code>
<code>float</code>	<code>System.Single</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>nint</code>	<code>System.IntPtr</code>
<code>nuint</code>	<code>System.UIntPtr</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>

The following table lists the C# built-in [reference](#) types:

C# TYPE KEYWORD	.NET TYPE
<code>object</code>	<code>System.Object</code>
<code>string</code>	<code>System.String</code>
<code>dynamic</code>	<code>System.Object</code>

In the preceding tables, each C# type keyword from the left column (except `nint` and `nuint` and `dynamic`) is an

alias for the corresponding .NET type. They are interchangeable. For example, the following declarations declare variables of the same type:

```
int a = 123;  
System.Int32 b = 123;
```

The `nint` and `nuint` types in the last two rows of the first table are native-sized integers. They are represented internally by the indicated .NET types, but in each case the keyword and the .NET type are not interchangeable. The compiler provides operations and conversions for `nint` and `nuint` as integer types that it doesn't provide for the pointer types `System.IntPtr` and `System.UIntPtr`. For more information, see [nint and nuint types](#).

The `void` keyword represents the absence of a type. You use it as the return type of a method that doesn't return a value.

See also

- [C# reference](#)
- [Default values of C# types](#)

Unmanaged types (C# reference)

4/6/2021 • 2 minutes to read • [Edit Online](#)

A type is an **unmanaged type** if it's any of the following types:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`
- Any `enum` type
- Any `pointer` type
- Any user-defined `struct` type that contains fields of unmanaged types only and, in C# 7.3 and earlier, is not a constructed type (a type that includes at least one type argument)

Beginning with C# 7.3, you can use the `unmanaged` constraint to specify that a type parameter is a non-pointer, non-nullable unmanaged type.

Beginning with C# 8.0, a *constructed* struct type that contains fields of unmanaged types only is also unmanaged, as the following example shows:

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

A generic struct may be the source of both unmanaged and not unmanaged constructed types. The preceding example defines a generic struct `Coords<T>` and presents the examples of unmanaged constructed types. The example of not an unmanaged type is `Coords<object>`. It's not unmanaged because it has the fields of the `object` type, which is not unmanaged. If you want *all* constructed types to be unmanaged types, use the `unmanaged` constraint in the definition of a generic struct:

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

C# language specification

For more information, see the [Pointer types](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Pointer types](#)
- [Memory and span-related types](#)
- [sizeof operator](#)
- [stackalloc](#)

Default values of C# types (C# reference)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The following table shows the default values of C# types:

TYPE	DEFAULT VALUE
Any reference type	<code>null</code>
Any built-in integral numeric type	0 (zero)
Any built-in floating-point numeric type	0 (zero)
<code>bool</code>	<code>false</code>
<code>char</code>	<code>'\0'</code> (U+0000)
<code>enum</code>	The value produced by the expression <code>(E)0</code> , where <code>E</code> is the enum identifier.
<code>struct</code>	The value produced by setting all value-type fields to their default values and all reference-type fields to <code>null</code> .
Any nullable value type	An instance for which the <code>HasValue</code> property is <code>false</code> and the <code>Value</code> property is undefined. That default value is also known as the <i>null</i> value of a nullable value type.

Use the `default` operator to produce the default value of a type, as the following example shows:

```
int a = default(int);
```

Beginning with C# 7.1, you can use the `default` literal to initialize a variable with the default value of its type:

```
int a = default;
```

For a value type, the implicit parameterless constructor also produces the default value of the type, as the following example shows:

```
var n = new System.Numerics.Complex();
Console.WriteLine(n); // output: (0, 0)
```

At run time, if the `System.Type` instance represents a value type, you can use the `Activator.CreateInstance(Type)` method to invoke the parameterless constructor to obtain the default value of the type.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Default values](#)

- [Default constructors](#)

See also

- [C# reference](#)
- [Constructors](#)

C# Keywords

4/7/2021 • 2 minutes to read • [Edit Online](#)

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program unless they include `@` as a prefix. For example, `@if` is a valid identifier, but `if` is not because `if` is a keyword.

The first table in this topic lists keywords that are reserved identifiers in any part of a C# program. The second table in this topic lists the contextual keywords in C#. Contextual keywords have special meaning only in a limited program context and can be used as identifiers outside that context. Generally, as new keywords are added to the C# language, they are added as contextual keywords in order to avoid breaking programs written in earlier versions.

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	record
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		

Contextual keywords

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#.

Some contextual keywords, such as `partial` and `where`, have special meanings in two or more contexts.

add	and	alias
ascending	async	await
by	descending	dynamic
equals	from	get
global	group	init
into	join	let
nameof	nint	not
notnull	nuint	on
or	orderby	partial (type)
partial (method)	remove	select
set	unmanaged (generic type constraint)	value
var	when (filter condition)	where (generic type constraint)
where (query clause)	with	yield

See also

- [C# reference](#)

Access Modifiers (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Access modifiers are keywords used to specify the declared accessibility of a member or a type. This section introduces the four access modifiers:

- `public`
- `protected`
- `internal`
- `private`

The following six accessibility levels can be specified using the access modifiers:

- `public`: Access is not restricted.
- `protected`: Access is limited to the containing class or types derived from the containing class.
- `internal`: Access is limited to the current assembly.
- `protected internal`: Access is limited to the current assembly or types derived from the containing class.
- `private`: Access is limited to the containing type.
- `private protected`: Access is limited to the containing class or types derived from the containing class within the current assembly.

This section also introduces the following:

- **Accessibility Levels**: Using the four access modifiers to declare six levels of accessibility.
- **Accessibility Domain**: Specifies where, in the program sections, a member can be referenced.
- **Restrictions on Using Accessibility Levels**: A summary of the restrictions on using declared accessibility levels.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Access Keywords](#)
- [Modifiers](#)

Accessibility Levels (C# Reference)

3/10/2021 • 2 minutes to read • [Edit Online](#)

Use the access modifiers, `public`, `protected`, `internal`, or `private`, to specify one of the following declared accessibility levels for members.

DECLARED ACCESSIBILITY	MEANING
<code>public</code>	Access is not restricted.
<code>protected</code>	Access is limited to the containing class or types derived from the containing class.
<code>internal</code>	Access is limited to the current assembly.
<code>protected internal</code>	Access is limited to the current assembly or types derived from the containing class.
<code>private</code>	Access is limited to the containing type.
<code>private protected</code>	Access is limited to the containing class or types derived from the containing class within the current assembly. Available since C# 7.2.

Only one access modifier is allowed for a member or type, except when you use the `protected internal` or `private protected` combinations.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

Top-level types, which are not nested in other types, can only have `internal` or `public` accessibility. The default accessibility for these types is `internal`.

Nested types, which are members of other types, can have declared accessibilities as indicated in the following table.

MEMBERS OF	DEFAULT MEMBER ACCESSIBILITY	ALLOWED DECLARED ACCESSIBILITY OF THE MEMBER
<code>enum</code>	<code>public</code>	None

MEMBERS OF	DEFAULT MEMBER ACCESSIBILITY	ALLOWED DECLARED ACCESSIBILITY OF THE MEMBER
class	private	public protected internal private protected internal private protected
interface	public	public protected internal private * protected internal private protected
struct	private	public internal private

* An `interface` member with `private` accessibility must have a default implementation.

The accessibility of a nested type depends on its [accessibility domain](#), which is determined by both the declared accessibility of the member and the accessibility domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Domain](#)
- [Restrictions on Using Accessibility Levels](#)
- [Access Modifiers](#)
- [public](#)
- [private](#)

- [protected](#)
- [internal](#)

Accessibility Domain (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The accessibility domain of a member specifies in which program sections a member can be referenced. If the member is nested within another type, its accessibility domain is determined by both the **accessibility level** of the member and the accessibility domain of the immediately containing type.

The accessibility domain of a top-level type is at least the program text of the project that it is declared in. That is, the domain includes all of the source files of this project. The accessibility domain of a nested type is at least the program text of the type in which it is declared. That is, the domain is the type body, which includes all nested types. The accessibility domain of a nested type never exceeds that of the containing type. These concepts are demonstrated in the following example.

Example

This example contains a top-level type, `T1`, and two nested classes, `M1` and `M2`. The classes contain fields that have different declared accessibilities. In the `Main` method, a comment follows each statement to indicate the accessibility domain of each member. Notice that the statements that try to reference the inaccessible members are commented out. If you want to see the compiler errors caused by referencing an inaccessible member, remove the comments one at a time.

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
```

```
static void Main()
{
    // Access is unlimited.
    T1.publicInt = 1;

    // Accessible only in current assembly.
    T1.internalInt = 2;

    // Error CS0122: inaccessible outside T1.
    // T1.privateInt = 3;

    // Access is unlimited.
    T1.M1.publicInt = 1;

    // Accessible only in current assembly.
    T1.M1.internalInt = 2;

    // Error CS0122: inaccessible outside M1.
    //     T1.M1.privateInt = 3;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.publicInt = 1;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.internalInt = 2;

    // Error CS0122: inaccessible outside M2.
    //     T1.M2.privateInt = 3;

    // Keep the console open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Restrictions on Using Accessibility Levels](#)
- [Access Modifiers](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

Restrictions on using accessibility levels (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

When you specify a type in a declaration, check whether the accessibility level of the type is dependent on the accessibility level of a member or of another type. For example, the direct base class must be at least as accessible as the derived class. The following declarations cause a compiler error because the base class `BaseClass` is less accessible than `MyClass`:

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

The following table summarizes the restrictions on declared accessibility levels.

CONTEXT	REMARKS
Classes	The direct base class of a class type must be at least as accessible as the class type itself.
Interfaces	The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
Delegates	The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
Constants	The type of a constant must be at least as accessible as the constant itself.
Fields	The type of a field must be at least as accessible as the field itself.
Methods	The return type and parameter types of a method must be at least as accessible as the method itself.
Properties	The type of a property must be at least as accessible as the property itself.
Events	The type of an event must be at least as accessible as the event itself.
Indexers	The type and parameter types of an indexer must be at least as accessible as the indexer itself.
Operators	The return type and parameter types of an operator must be at least as accessible as the operator itself.
Constructors	The parameter types of a constructor must be at least as accessible as the constructor itself.

Example

The following example contains erroneous declarations of different types. The comment following each declaration indicates the expected compiler error.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.WriteLine("Compiled successfully");
    }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Domain](#)
- [Accessibility Levels](#)
- [Access Modifiers](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

internal (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `internal` keyword is an [access modifier](#) for types and type members.

This page covers `internal` access. The `internal` keyword is also part of the `protected internal` access modifier.

Internal types or members are accessible only within files in the same assembly, as in this example:

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

For a comparison of `internal` with the other access modifiers, see [Accessibility Levels](#) and [Access Modifiers](#).

For more information about assemblies, see [Assemblies in .NET](#).

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide `control` and `Form` classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

Example

This example contains two files, `Assembly1.cs` and `Assembly1_a.cs`. The first file contains an internal base class, `BaseClass`. In the second file, an attempt to instantiate `BaseClass` will produce an error.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
    public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass();    // CS0122
    }
}
```

Example

In this example, use the same files you used in example 1, and change the accessibility level of `BaseClass` to `public`. Also change the accessibility level of the member `intM` to `internal`. In this case, you can instantiate the class, but you cannot access the internal member.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
    internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // Ok.
        BaseClass.intM = 444; // CS0117
    }
}
```

C# Language Specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [protected](#)

private (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `private` keyword is a member access modifier.

This page covers `private` access. The `private` keyword is also part of the `private protected` access modifier.

Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared, as in this example:

```
class Employee
{
    private int i;
    double d; // private access by default
}
```

Nested types in the same body can also access those private members.

It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

For a comparison of `private` with the other access modifiers, see [Accessibility Levels](#) and [Access Modifiers](#).

Example

In this example, the `Employee` class contains two private data members, `name` and `salary`. As private members, they cannot be accessed except by member methods. Public methods named `GetName` and `Salary` are added to allow controlled access to the private members. The `name` member is accessed by way of a public method, and the `salary` member is accessed by way of a public read-only property. (See [Properties](#) for more information.)

```
class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //     string n = e.name;
        //     double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

C# language specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [protected](#)
- [internal](#)

protected (C# Reference)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The `protected` keyword is a member access modifier.

NOTE

This page covers `protected` access. The `protected` keyword is also part of the `protected internal` and `private protected` access modifiers.

A protected member is accessible within its class and by derived class instances.

For a comparison of `protected` with the other access modifiers, see [Accessibility Levels](#).

Example

A protected member of a base class is accessible in a derived class only if the access occurs through the derived class type. For example, consider the following code segment:

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

The statement `a.x = 10` generates an error because it is made within the static method `Main`, and not an instance of class `B`.

Struct members cannot be protected because the struct cannot be inherited.

Example

In this example, the class `DerivedPoint` is derived from `Point`. Therefore, you can access the protected members of the base class directly from the derived class.

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

If you change the access levels of `x` and `y` to [private](#), the compiler will issue the error messages:

`'Point.y' is inaccessible due to its protection level.`

`'Point.x' is inaccessible due to its protection level.`

C# language specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [internal](#)
- [Security concerns for internal virtual keywords](#)

public (C# Reference)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The `public` keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members, as in this example:

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

See [Access Modifiers](#) and [Accessibility Levels](#) for more information.

Example

In the following example, two classes are declared, `PointTest` and `Program`. The public members `x` and `y` of `PointTest` are accessed directly from `Program`.

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

If you change the `public` access level to `private` or `protected`, you will get the error message:

'`PointTest.y`' is inaccessible due to its protection level.

C# language specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Access Modifiers](#)
- [C# Keywords](#)
- [Access Modifiers](#)

- Accessibility Levels
- Modifiers
- private
- protected
- internal

protected internal (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `protected internal` keyword combination is a member access modifier. A protected internal member is accessible from the current assembly or from types that are derived from the containing class. For a comparison of `protected internal` with the other access modifiers, see [Accessibility Levels](#).

Example

A protected internal member of a base class is accessible from any type within its containing assembly. It is also accessible in a derived class located in another assembly only if the access occurs through a variable of the derived class type. For example, consider the following code segment:

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and another class, `TestAccess`. `BaseClass` owns a protected internal member, `myValue`, which is accessed by the `TestAccess` type. In the second file, an attempt to access `myValue` through an instance of `BaseClass` will produce an error, while an access to this member through an instance of a derived class, `DerivedClass` will succeed.

Struct members cannot be `protected internal` because the struct cannot be inherited.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [internal](#)
- [Security concerns for internal virtual keywords](#)

private protected (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `private protected` keyword combination is a member access modifier. A private protected member is accessible by types derived from the containing class, but only within its containing assembly. For a comparison of `private protected` with the other access modifiers, see [Accessibility Levels](#).

NOTE

The `private protected` access modifier is valid in C# version 7.2 and later.

Example

A private protected member of a base class is accessible from derived types in its containing assembly only if the static type of the variable is the derived class type. For example, consider the following code segment:

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and a type derived from it, `DerivedClass1`. `BaseClass` owns a private protected member, `myValue`, which `DerivedClass1` tries to access in two ways. The first attempt to access `myValue` through an instance of `BaseClass` will produce an error. However, the attempt to use it as an inherited member in `DerivedClass1` will succeed.

In the second file, an attempt to access `myValue` as an inherited member of `DerivedClass2` will produce an error, as it is only accessible by derived types in Assembly1.

If `Assembly1.cs` contains an `InternalsVisibleToAttribute` that names `Assembly2`, the derived class `DerivedClass1` will have access to `private protected` members declared in `BaseClass`. `InternalsVisibleTo` makes `private protected` members visible to derived classes in other assemblies.

Struct members cannot be `private protected` because the struct cannot be inherited.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [internal](#)
- [Security concerns for internal virtual keywords](#)

abstract (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. The `abstract` modifier can be used with classes, methods, properties, indexers, and events. Use the `abstract` modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as `abstract` must be implemented by non-abstract classes that derive from the abstract class.

Example

In this example, the class `Square` must provide an implementation of `GetArea` because it derives from `Shape`:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the `sealed` modifier because the two modifiers have opposite meanings. The `sealed` modifier prevents a class from being inherited and the `abstract` modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the `abstract` modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.

- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({}) following the signature. For example:

```
public abstract void MyMethod();
```

The implementation is provided by a method `override`, which is a member of a non-abstract class.

- It is an error to use the `static` or `virtual` modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `abstract` modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the `override` modifier.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members](#).

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Example

In this example, the class `DerivedClass` is derived from an abstract class `BaseClass`. The abstract class contains an abstract method, `AbstractMethod`, and two abstract properties, `X` and `Y`.

```

abstract class BaseClass // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod(); // Abstract method
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161

```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

```
BaseClass bc = new BaseClass(); // Error
```

You will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Modifiers](#)
- [virtual](#)
- [override](#)

- C# Keywords

async (C# Reference)

11/2/2020 • 4 minutes to read • [Edit Online](#)

Use the `async` modifier to specify that a method, [lambda expression](#), or [anonymous method](#) is asynchronous. If you use this modifier on a method or expression, it's referred to as an *async method*. The following example defines an `async` method named `ExampleMethodAsync`:

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

If you're new to asynchronous programming or do not understand how an `async` method uses the `await` operator to do potentially long-running work without blocking the caller's thread, read the introduction in [Asynchronous programming with `async` and `await`](#). The following code is found inside an `async` method and calls the [HttpClient.GetStringAsync](#) method:

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

An `async` method runs synchronously until it reaches its first `await` expression, at which point the method is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method, as the example in the next section shows.

If the method that the `async` keyword modifies doesn't contain an `await` expression or statement, the method executes synchronously. A compiler warning alerts you to any `async` methods that don't contain `await` statements, because that situation might indicate an error. See [Compiler Warning \(level 1\) CS4014](#).

The `async` keyword is contextual in that it's a keyword only when it modifies a method, a lambda expression, or an anonymous method. In all other contexts, it's interpreted as an identifier.

Example

The following example shows the structure and flow of control between an `async` event handler, `StartButton_Click`, and an `async` method, `ExampleMethodAsync`. The result from the `async` method is the number of characters of a web page. The code is suitable for a Windows Presentation Foundation (WPF) app or Windows Store app that you create in Visual Studio; see the code comments for setting up the app.

You can run this code in Visual Studio as a Windows Presentation Foundation (WPF) app or a Windows Store app. You need a Button control named `StartButton` and a Textbox control named `ResultsTextBox`. Remember to set the names and handler so that you have something like this:

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"
       Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
        Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

To run the code as a WPF app:

- Paste this code into the `MainWindow` class in `MainWindow.xaml.cs`.
- Add a reference to `System.Net.Http`.

- Add a `using` directive for `System.Net.Http`.

To run the code as a Windows Store app:

- Paste this code into the `MainPage` class in `MainPage.xaml.cs`.

- Add `using` directives for `System.Net.Http` and `System.Threading.Tasks`.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

IMPORTANT

For more information about tasks and the code that executes while waiting for a task, see [Asynchronous programming with `async` and `await`](#). For a full console example that uses similar elements, see [Process asynchronous tasks as they complete \(C#\)](#).

Return Types

An `async` method can have the following return types:

- `Task`
- `Task<TResult>`
- `void`. `async void` methods are generally discouraged for code other than event handlers because callers cannot `await` those methods and must implement a different mechanism to report successful completion or error conditions.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The `System.Threading.Tasks.ValueTask<TResult>` type is one such implementation. It is available by adding the

NuGet package `System.Threading.Tasks.Extensions`.

The `async` method can't declare any `in`, `ref` or `out` parameters, nor can it have a [reference return value](#), but it can call methods that have such parameters.

You specify `Task<TResult>` as the return type of an `async` method if the `return` statement of the method specifies an operand of type `TResult`. You use `Task` if no meaningful value is returned when the method is completed. That is, a call to the method returns a `Task`, but when the `Task` is completed, any `await` expression that's awaiting the `Task` evaluates to `void`.

You use the `void` return type primarily to define event handlers, which require that return type. The caller of a `void`-returning `async` method can't await it and can't catch exceptions that the method throws.

Starting with C# 7.0, you return another type, typically a value type, that has a `GetAwaiter` method to minimize memory allocations in performance-critical sections of code.

For more information and examples, see [Async Return Types](#).

See also

- [AsyncStateMachineAttribute](#)
- [await](#)
- [Asynchronous programming with async and await](#)
- [Process asynchronous tasks as they complete](#)

const (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

You use the `const` keyword to declare a constant field or a constant local. Constant fields and locals aren't variables and may not be modified. Constants can be numbers, Boolean values, strings, or a null reference. Don't create a constant to represent information that you expect to change at any time. For example, don't use a constant field to store the price of a service, a product version number, or the brand name of a company. These values can change over time, and because compilers propagate constants, other code compiled with your libraries will have to be recompiled to see the changes. See also the `readonly` keyword. For example:

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Remarks

The type of a constant declaration specifies the type of the members that the declaration introduces. The initializer of a constant local or a constant field must be a constant expression that can be implicitly converted to the target type.

A constant expression is an expression that can be fully evaluated at compile time. Therefore, the only possible values for constants of reference types are `string` and a null reference.

The constant declaration can declare multiple constants, such as:

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

The `static` modifier is not allowed in a constant declaration.

A constant can participate in a constant expression, as follows:

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

NOTE

The `readonly` keyword differs from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, although a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants, as in this line:

```
public static readonly uint l1 = (uint)DateTime.Now.Ticks;
```

Example

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
   x = 11, y = 22
   C1 = 5, C2 = 10
*/
```

Example

This example demonstrates how to use constants as local variables.

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [readonly](#)

event (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `event` keyword is used to declare an event in a publisher class.

Example

The following example shows how to declare and raise an event that uses `EventHandler` as the underlying delegate type. For the complete code example that also shows how to use the generic `EventHandler<TEventArgs>` delegate type and how to subscribe to an event and create an event handler method, see [How to publish events that conform to .NET Guidelines](#).

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Events are a special kind of multicast delegate that can only be invoked from within the class or struct where they are declared (the publisher class). If other classes or structs subscribe to the event, their event handler methods will be called when the publisher class raises the event. For more information and code examples, see [Events and Delegates](#).

Events can be marked as `public`, `private`, `protected`, `internal`, `protected internal`, or `private protected`. These access modifiers define how users of the class can access the event. For more information, see [Access Modifiers](#).

Keywords and events

The following keywords apply to events.

KEYWORD	DESCRIPTION	FOR MORE INFORMATION
<code>static</code>	Makes the event available to callers at any time, even if no instance of the class exists.	Static Classes and Static Class Members

KEYWORD	DESCRIPTION	FOR MORE INFORMATION
<code>virtual</code>	Allows derived classes to override the event behavior by using the <code>override</code> keyword.	Inheritance
<code>sealed</code>	Specifies that for derived classes it is no longer virtual.	
<code>abstract</code>	The compiler will not generate the <code>add</code> and <code>remove</code> event accessor blocks and therefore derived classes must provide their own implementation.	

An event may be declared as a static event by using the `static` keyword. This makes the event available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members](#).

An event can be marked as a virtual event by using the `virtual` keyword. This enables derived classes to override the event behavior by using the `override` keyword. For more information, see [Inheritance](#). An event overriding a virtual event can also be `sealed`, which specifies that for derived classes it is no longer virtual. Lastly, an event can be declared `abstract`, which means that the compiler will not generate the `add` and `remove` event accessor blocks. Therefore derived classes must provide their own implementation.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [add](#)
- [remove](#)
- [Modifiers](#)
- [How to combine delegates \(Multicast Delegates\)](#)

extern (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `extern` modifier is used to declare a method that is implemented externally. A common use of the `extern` modifier is with the `[DllImport]` attribute when you are using Interop services to call into unmanaged code. In this case, the method must also be declared as `static`, as shown in the following example:

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

The `extern` keyword can also define an external assembly alias, which makes it possible to reference different versions of the same component from within a single assembly. For more information, see [extern alias](#).

It is an error to use the `abstract` and `extern` modifiers together to modify the same member. Using the `extern` modifier means that the method is implemented outside the C# code, whereas using the `abstract` modifier means that the method implementation is not provided in the class.

The `extern` keyword has more limited uses in C# than in C++. To compare the C# keyword with the C++ keyword, see [Using extern to Specify Linkage in the C++ Language Reference](#).

Example 1

In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox` method imported from the User32.dll library.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Example 2

This example illustrates a C# program that calls into a C library (a native DLL).

1. Create the following C file and name it `cmdll.c`:

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
    return i*10;
}
```

2. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cmdll.c` file by typing `cl -LD cmdll.c` at the command prompt.

3. In the same directory, create the following C# file and name it `cm.cs`:

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
    public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cm.cs` file by typing:

```
csc cm.cs (for the x64 command prompt) —or— csc -platform:x86 cm.cs (for the x32 command prompt)
```

This will create the executable file `cm.exe`.

5. Run `cm.exe`. The `SampleMethod` method passes the value 5 to the DLL file, which returns the value multiplied by 10. The program produces the following output:

```
SampleMethod() returns 50.
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [System.Runtime.InteropServices.DllImportAttribute](#)
- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)

in (Generic Modifier) (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

For generic type parameters, the `in` keyword specifies that the type parameter is contravariant. You can use the `in` keyword in generic interfaces and delegates.

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement contravariant interfaces and implicit conversion of delegate types. Covariance and contravariance in generic type parameters are supported for reference types, but they are not supported for value types.

A type can be declared contravariant in a generic interface or delegate only if it defines the type of a method's parameters and not of a method's return type. `In`, `ref`, and `out` parameters must be invariant, meaning they are neither covariant or contravariant.

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, in the `IComparer<T>` interface, type `T` is contravariant, you can assign an object of the `IComparer<Person>` type to an object of the `IComparer<Employee>` type without using any special conversion methods if `Employee` inherits `Person`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

For more information, see [Covariance and Contravariance](#).

Contravariant generic interface

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

```
// Contravariant interface.  
interface IContravariant<in A> { }  
  
// Extending contravariant interface.  
interface IExtContravariant<in A> : IContravariant<A> { }  
  
// Implementing contravariant interface.  
class Sample<A> : IContravariant<A> { }  
  
class Program  
{  
    static void Test()  
    {  
        IContravariant<Object> iobj = new Sample<Object>();  
        IContravariant<String> istr = new Sample<String>();  
  
        // You can assign iobj to istr because  
        // the IContravariant interface is contravariant.  
        istr = iobj;  
    }  
}
```

Contravariant generic delegate

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also

shows how you can implicitly convert a delegate type.

```
// Contravariant delegate.  
public delegate void DContravariant<in A>(A argument);  
  
// Methods that match the delegate signature.  
public static void SampleControl(Control control)  
{ }  
public static void SampleButton(Button button)  
{ }  
  
public void Test()  
{  
  
    // Instantiating the delegates with the methods.  
    DContravariant<Control> dControl = SampleControl;  
    DContravariant<Button> dButton = SampleButton;  
  
    // You can assign dControl to dButton  
    // because the DContravariant delegate is contravariant.  
    dButton = dControl;  
  
    // Invoke the delegate.  
    dButton(new Button());  
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [out](#)
- [Covariance and Contravariance](#)
- [Modifiers](#)

new modifier (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

When used as a declaration modifier, the `new` keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. Although you can hide members without using the `new` modifier, you get a compiler warning. If you use `new` to explicitly hide a member, it suppresses this warning.

You can also use the `new` keyword to [create an instance of a type](#) or as a [generic type constraint](#).

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the `new` keyword. For example:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

In this example, `BaseC.Invoke` is hidden by `DerivedC.Invoke`. The field `x` is not affected because it is not hidden by a similar name.

Name hiding through inheritance takes one of the following forms:

- Generally, a constant, field, property, or type that is introduced in a class or struct hides all base class members that share its name. There are special cases. For example, if you declare a new field with name `N` to have a type that is not invocable, and a base type declares `N` to be a method, the new field does not hide the base declaration in invocation syntax. For more information, see the [Member lookup](#) section of the [C# language specification](#).
- A method introduced in a class or struct hides properties, fields, and types that share that name in the base class. It also hides all base class methods that have the same signature.
- An indexer introduced in a class or struct hides all base class indexers that have the same signature.

It is an error to use both `new` and `override` on the same member, because the two modifiers have mutually exclusive meanings. The `new` modifier creates a new member with the same name and causes the original member to become hidden. The `override` modifier extends the implementation for an inherited member.

Using the `new` modifier in a declaration that does not hide an inherited member generates a warning.

Example

In this example, a base class, `BaseC`, and a derived class, `DerivedC`, use the same field name `x`, which hides the value of the inherited field. The example demonstrates the use of the `new` modifier. It also demonstrates how to access the hidden members of the base class by using their fully qualified names.

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

Example

In this example, a nested class hides a class that has the same name in the base class. The example demonstrates how to use the `new` modifier to eliminate the warning message and how to access the hidden class members by using their fully qualified names.

```

public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1 = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/

```

If you remove the `new` modifier, the program will still compile and run, but you will get the following warning:

The keyword new is required on 'MyDerivedC.x' because it hides inherited member ' MyBaseC.x'.

C# language specification

For more information, see [The new modifier](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [Versioning with the Override and New Keywords](#)
- [Knowing When to Use Override and New Keywords](#)

out (generic modifier) (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

For generic type parameters, the `out` keyword specifies that the type parameter is covariant. You can use the `out` keyword in generic interfaces and delegates.

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement covariant interfaces and implicit conversion of delegate types. Covariance and contravariance are supported for reference types, but they are not supported for value types.

An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter. For example, because in .NET Framework 4, in `IEnumerable<T>`, type `T` is covariant, you can assign an object of the `IEnumerable<OF String>` type to an object of the `IEnumerable<OF Object>` type without using any special conversion methods.

A covariant delegate can be assigned another delegate of the same type, but with a more derived generic type parameter.

For more information, see [Covariance and Contravariance](#).

Example - covariant generic interface

The following example shows how to declare, extend, and implement a covariant generic interface. It also shows how to use implicit conversion for classes that implement a covariant interface.

```
// Covariant interface.  
interface ICovariant<out R> { }  
  
// Extending covariant interface.  
interface IExtCovariant<out R> : ICovariant<R> { }  
  
// Implementing covariant interface.  
class Sample<R> : ICovariant<R> { }  
  
class Program  
{  
    static void Test()  
    {  
        ICovariant<Object> iobj = new Sample<Object>();  
        ICovariant<String> istr = new Sample<String>();  
  
        // You can assign istr to iobj because  
        // the ICovariant interface is covariant.  
        iobj = istr;  
    }  
}
```

In a generic interface, a type parameter can be declared covariant if it satisfies the following conditions:

- The type parameter is used only as a return type of interface methods and not used as a type of method arguments.

NOTE

There is one exception to this rule. If in a covariant interface you have a contravariant generic delegate as a method parameter, you can use the covariant type as a generic type parameter for this delegate. For more information about covariant and contravariant generic delegates, see [Variance in Delegates](#) and [Using Variance for Func and Action Generic Delegates](#).

- The type parameter is not used as a generic constraint for the interface methods.

Example - covariant generic delegate

The following example shows how to declare, instantiate, and invoke a covariant generic delegate. It also shows how to implicitly convert delegate types.

```
// Covariant delegate.  
public delegate R DCovariant<out R>();  
  
// Methods that match the delegate signature.  
public static Control SampleControl()  
{ return new Control(); }  
  
public static Button SampleButton()  
{ return new Button(); }  
  
public void Test()  
{  
    // Instantiate the delegates with the methods.  
    DCovariant<Control> dControl = SampleControl;  
    DCovariant<Button> dButton = SampleButton;  
  
    // You can assign dButton to dControl  
    // because the DCovariant delegate is covariant.  
    dControl = dButton;  
  
    // Invoke the delegate.  
    dControl();  
}
```

In a generic delegate, a type can be declared covariant if it is used only as a method return type and not used for method arguments.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Variance in Generic Interfaces](#)
- [in](#)
- [Modifiers](#)

override (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `override` modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

In the following example, the `Square` class must provide an overridden implementation of `GetArea` because `GetArea` is inherited from the abstract `Shape` class:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

An `override` method provides a new implementation of the method inherited from a base class. The method that is overridden by an `override` declaration is known as the overridden base method. An `override` method must have the same signature as the overridden base method. Beginning with C# 9.0, `override` methods support covariant return types. In particular, the return type of an `override` method can derive from the return type of the corresponding base method. In C# 8.0 and earlier, the return types of an `override` method and the overridden base method must be the same.

You cannot override a non-virtual or static method. The overridden base method must be `virtual`, `abstract`, or `override`.

An `override` declaration cannot change the accessibility of the `virtual` method. Both the `override` method and the `virtual` method must have the same [access level modifier](#).

You cannot use the `new`, `static`, or `virtual` modifiers to modify an `override` method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property. Beginning with C# 9.0, read-only overriding properties support covariant return types. The overridden property must be `virtual`, `abstract`, or `override`.

For more information about how to use the `override` keyword, see [Versioning with the Override and New Keywords](#) and [Knowing when to use Override and New Keywords](#). For information about inheritance, see [Inheritance](#).

Example

This example defines a base class named `Employee`, and a derived class named `SalesEmployee`. The `SalesEmployee` class includes an extra field, `salesbonus`, and overrides the method `calculatePay` in order to take it into account.

```

class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                             decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice",
                                         1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.name} earned: {employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.name} earned: {employee2.CalculatePay()}");
    }
}
/*
Output:
Employee1 Alice earned: 1500
Employee2 Bob earned: 1200
*/

```

C# language specification

For more information, see the [Override methods](#) section of the [C# language specification](#).

For more information about covariant return types, see the [feature proposal note](#).

See also

- [C# reference](#)
- [Inheritance](#)
- [C# keywords](#)
- [Modifiers](#)
- [abstract](#)
- [virtual](#)
- [new \(modifier\)](#)
- [Polymorphism](#)

readonly (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `readonly` keyword is a modifier that can be used in four contexts:

- In a [field declaration](#), `readonly` indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A `readonly` field can be assigned and reassigned multiple times within the field declaration and constructor.
A `readonly` field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:
 - Because value types directly contain their data, a field that is a `readonly` value type is immutable.
 - Because reference types contain a reference to their data, a field that is a `readonly` reference type must always refer to the same object. That object isn't immutable. The `readonly` modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.

WARNING

An externally visible type that contains an externally visible read-only field that is a mutable reference type may be a security vulnerability and may trigger warning [CA2104](#) : "Do not declare read only mutable reference types."

- In a `readonly struct` type definition, `readonly` indicates that the structure type is immutable. For more information, see the [readonly struct](#) section of the [Structure types](#) article.
- In an instance member declaration within a structure type, `readonly` indicates that an instance member doesn't modify the state of the structure. For more information, see the [readonly instance members](#) section of the [Structure types](#) article.
- In a `ref readonly` method return, the `readonly` modifier indicates that method returns a reference and writes aren't allowed to that reference.

The `readonly struct` and `ref readonly` contexts were added in C# 7.2. `readonly` struct members were added in C# 8.0.

Readonly field example

In this example, the value of the field `year` can't be changed in the method `ChangeYear`, even though it's assigned a value in the class constructor:

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

You can assign a value to a `readonly` field only in the following contexts:

- When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```

- In an instance constructor of the class that contains the instance field declaration.
- In the static constructor of the class that contains the static field declaration.

These constructor contexts are also the only contexts in which it's valid to pass a `readonly` field as an `out` or `ref` parameter.

NOTE

The `readonly` keyword is different from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be assigned multiple times in the field declaration and in any constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, while a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants as in the following example:

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```

public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32); // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55; // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
}
/*
Output:
p1: x=11, y=21, z=32
p2: x=55, y=25, z=24
*/
}

```

In the preceding example, if you use a statement like the following example:

```
p2.y = 66; // Error
```

you'll get the compiler error message:

A readonly field cannot be assigned to (except in a constructor or a variable initializer)

Ref readonly return example

The `readonly` modifier on a `ref return` indicates that the returned reference can't be modified. The following example returns a reference to the origin. It uses the `readonly` modifier to indicate that callers can't modify the origin:

```

private static readonly SamplePoint origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref origin;

```

The type returned doesn't need to be a `readonly struct`. Any type that can be returned by `ref` can be returned by `ref readonly`.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

You can also see the language specification proposals:

- [readonly ref and readonly struct](#)
- [readonly struct members](#)

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [const](#)
- [Fields](#)

sealed (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it. In the following example, class `B` inherits from class `A`, but no class can inherit from class `B`.

```
class A {}
sealed class B : A {}
```

You can also use the `sealed` modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, `Z` inherits from `Y` but `Z` cannot override the virtual function `F` that is declared in `X` and sealed in `Y`.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as `virtual`.

It is an error to use the `abstract` modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the `sealed` modifier must always be used with `override`.

Because structs are implicitly sealed, they cannot be inherited.

For more information, see [Inheritance](#).

For more examples, see [Abstract and Sealed Classes and Class Members](#).

Example

```

sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150

```

In the previous example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

The result is an error message:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Remarks

To determine whether to seal a class, method, or property, you should generally consider the following two points:

- The potential benefits that deriving classes might gain through the ability to customize your class.
- The potential that deriving classes could modify your classes in such a way that they would no longer work correctly or as expected.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Static Classes and Static Class Members](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Access Modifiers](#)
- [Modifiers](#)
- [override](#)
- [virtual](#)

static (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

This page covers the `static` modifier keyword. The `static` keyword is also part of the `using static` directive.

Use the `static` modifier to declare a static member, which belongs to the type itself rather than to a specific object. The `static` modifier can be used to declare `static` classes. In classes, interfaces, and structs, you may add the `static` modifier to fields, methods, properties, operators, events, and constructors. The `static` modifier can't be used with indexers or finalizers. For more information, see [Static Classes and Static Class Members](#).

Beginning with C# 8.0, you can add the `static` modifier to a [local function](#). A static local function can't capture local variables or instance state.

Beginning with C# 9.0, you can add the `static` modifier to a [lambda expression](#) or [anonymous method](#). A static lambda or anonymous method can't capture local variables or instance state.

Example - static class

The following class is declared as `static` and contains only `static` methods:

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/ }
}
```

A constant or type declaration is implicitly a `static` member. A `static` member can't be referenced through an instance. Instead, it's referenced through the type name. For example, consider the following class:

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

To refer to the `static` member `x`, use the fully qualified name, `MyBaseC.MyStruct.x`, unless the member is accessible from the same scope:

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

While an instance of a class contains a separate copy of all instance fields of the class, there's only one copy of each `static` field.

It isn't possible to use `this` to reference `static` methods or property accessors.

If the `static` keyword is applied to a class, all the members of the class must be `static`.

Classes, interfaces, and `static` classes may have `static` constructors. A `static` constructor is called at some point between when the program starts and the class is instantiated.

NOTE

The `static` keyword has more limited uses than in C++. To compare with the C++ keyword, see [Storage classes \(C++\)](#).

To demonstrate `static` members, consider a class that represents a company employee. Assume that the class contains a method to count employees and a field to store the number of employees. Both the method and the field don't belong to any one employee instance. Instead, they belong to the class of employees as a whole. They should be declared as `static` members of the class.

Example - static field and method

This example reads the name and ID of a new employee, increments the employee counter by one, and displays the information for the new employee and the new number of employees. This program reads the current number of employees from the keyboard.

```

public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.WriteLine("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID: {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
*
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID: AF643G
New Number of Employees: 16
*/

```

Example - static initialization

This example shows that you can initialize a `static` field by using another `static` field that is not yet declared. The results will be undefined until you explicitly assign a value to the `static` field.

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
0
5
99
*/
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [using static directive](#)
- [Static Classes and Static Class Members](#)

unsafe (C# Reference)

4/6/2021 • 2 minutes to read • [Edit Online](#)

The `unsafe` keyword denotes an unsafe context, which is required for any operation involving pointers. For more information, see [Unsafe Code and Pointers](#).

You can use the `unsafe` modifier in the declaration of a type or a member. The entire textual extent of the type or member is therefore considered an unsafe context. For example, the following is a method declared with the `unsafe` modifier:

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

The scope of the unsafe context extends from the parameter list to the end of the method, so pointers can also be used in the parameter list:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

You can also use an unsafe block to enable the use of an unsafe code inside this block. For example:

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

To compile unsafe code, you must specify the [AllowUnsafeBlocks](#) compiler option. Unsafe code is not verifiable by the common language runtime.

Example

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

C# language specification

For more information, see [Unsafe code](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [fixed Statement](#)
- [Unsafe Code and Pointers](#)
- [Fixed Size Buffers](#)

virtual (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `virtual` keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

```
public virtual double Area()
{
    return x * y;
}
```

The implementation of a virtual member can be changed by an [overriding member](#) in a derived class. For more information about how to use the `virtual` keyword, see [Versioning with the Override and New Keywords](#) and [Knowing When to Use Override and New Keywords](#).

Remarks

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the `virtual` modifier with the `static`, `abstract`, `private`, or `override` modifiers. The following example shows a virtual property:

```

class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}

```

Virtual properties behave like virtual methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `virtual` modifier on a static property.
- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the `override` modifier.

Example

In this example, the `Shape` class contains the two coordinates `x`, `y`, and the `Area()` virtual method. Different shape classes such as `Circle`, `Cylinder`, and `Sphere` inherit the `Shape` class, and the surface area is calculated for each figure. Each derived class has its own override implementation of `Area()`.

Notice that the inherited classes `Circle`, `Sphere`, and `Cylinder` all use constructors that initialize the base class, as shown in the following declaration.

```
public Cylinder(double r, double h): base(r, h) {}
```

The following program calculates and displays the appropriate area for each figure by invoking the appropriate implementation of the `Area()` method, according to the object that is associated with the method.

```

class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;

        public Shape()
        {
        }

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

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * x * x;
        }
    }

    class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * x * x + 2 * PI * x * y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle    = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
    }
}

```

```
        Console.WriteLine("Area of Cylinder = {0:F2}", 1.Area());
    }
}
/*
Output:
Area of Circle    = 28.27
Area of Sphere    = 113.10
Area of Cylinder = 150.80
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Polymorphism](#)
- [abstract](#)
- [override](#)
- [new \(modifier\)](#)

volatile (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `volatile` keyword indicates that a field might be modified by multiple threads that are executing at the same time. The compiler, the runtime system, and even hardware may rearrange reads and writes to memory locations for performance reasons. Fields that are declared `volatile` are not subject to these optimizations. Adding the `volatile` modifier ensures that all threads will observe volatile writes performed by any other thread in the order in which they were performed. There is no guarantee of a single total ordering of volatile writes as seen from all threads of execution.

The `volatile` keyword can be applied to fields of these types:

- Reference types.
- Pointer types (in an unsafe context). Note that although the pointer itself can be volatile, the object that it points to cannot. In other words, you cannot declare a "pointer to volatile."
- Simple types such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`.
- An `enum` type with one of the following base types: `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.
- Generic type parameters known to be reference types.
- `IntPtr` and `UIntPtr`.

Other types, including `double` and `long`, cannot be marked `volatile` because reads and writes to fields of those types cannot be guaranteed to be atomic. To protect multi-threaded access to those types of fields, use the `Interlocked` class members or protect access using the `lock` statement.

The `volatile` keyword can only be applied to fields of a `class` or `struct`. Local variables cannot be declared `volatile`.

Example

The following example shows how to declare a public field variable as `volatile`.

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int _i)
    {
        sharedStorage = _i;
    }
}
```

The following example demonstrates how an auxiliary or worker thread can be created and used to perform processing in parallel with that of the primary thread. For more information about multithreading, see [Managed Threading](#).

```

public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}

```

With the `volatile` modifier added to the declaration of `_shouldStop` in place, you'll always get the same results (similar to the excerpt shown in the preceding code). However, without that modifier on the `_shouldStop` member, the behavior is unpredictable. The `DoWork` method may optimize the member access, resulting in reading stale data. Because of the nature of multi-threaded programming, the number of stale reads is unpredictable. Different runs of the program will produce somewhat different results.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for

C# syntax and usage.

See also

- [C# language specification: volatile keyword](#)
- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [lock statement](#)
- [Interlocked](#)

Statement keywords (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Statements are program instructions. Except as described in the topics referenced in the following table, statements are executed in sequence. The following table lists the C# statement keywords. For more information about statements that are not expressed with any keyword, see [Statements](#).

CATEGORY	C# KEYWORDS
Selection statements	if , else , switch , case
Iteration statements	do , for , foreach , in , while
Jump statements	break , continue , default , goto , return , yield
Exception handling statements	throw , try-catch , try-finally , try-catch-finally
Checked and unchecked	checked , unchecked
fixed statement	fixed
lock statement	lock

See also

- [C# Reference](#)
- [Statements](#)
- [C# Keywords](#)

if-else (C# Reference)

11/2/2020 • 5 minutes to read • [Edit Online](#)

An `if` statement identifies which statement to run based on the value of a Boolean expression. In the following example, the `bool` variable `condition` is set to `true` and then checked in the `if` statement. The output is `The variable is set to true.`.

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

You can run the examples in this topic by placing them in the `Main` method of a console app.

An `if` statement in C# can take two forms, as the following example shows.

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

In an `if-else` statement, if `condition` evaluates to true, the `then-statement` runs. If `condition` is false, the `else-statement` runs. Because `condition` can't be simultaneously true and false, the `then-statement` and the `else-statement` of an `if-else` statement can never both run. After the `then-statement` or the `else-statement` runs, control is transferred to the next statement after the `if` statement.

In an `if` statement that doesn't include an `else` statement, if `condition` is true, the `then-statement` runs. If `condition` is false, control is transferred to the next statement after the `if` statement.

Both the `then-statement` and the `else-statement` can consist of a single statement or multiple statements that are enclosed in braces (`{}`). For a single statement, the braces are optional but recommended.

The statement or statements in the `then-statement` and the `else-statement` can be of any kind, including another `if` statement nested inside the original `if` statement. In nested `if` statements, each `else` clause belongs to the last `if` that doesn't have a corresponding `else`. In the following example, `Result1` appears if both `m > 10` and `n > 20` evaluate to true. If `m > 10` is true but `n > 20` is false, `Result2` appears.

```
// Try with m = 12 and then with m = 8.  
int m = 12;  
int n = 18;  
  
if (m > 10)  
    if (n > 20)  
    {  
        Console.WriteLine("Result1");  
    }  
    else  
    {  
        Console.WriteLine("Result2");  
    }
```

If, instead, you want `Result2` to appear when `(m > 10)` is false, you can specify that association by using braces to establish the start and end of the nested `if` statement, as the following example shows.

```
// Try with m = 12 and then with m = 8.  
if (m > 10)  
{  
    if (n > 20)  
        Console.WriteLine("Result1");  
}  
else  
{  
    Console.WriteLine("Result2");  
}
```

`Result2` appears if the condition `(m > 10)` evaluates to false.

Example

In the following example, you enter a character from the keyboard, and the program uses a nested `if` statement to determine whether the input character is an alphabetic character. If the input character is an alphabetic character, the program checks whether the input character is lowercase or uppercase. A message appears for each case.

```
Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.ToLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character.");
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.
```

Example

You can also nest an `if` statement inside an `else` block, as the following partial code shows. The example nests `if` statements inside two `else` blocks and one `then` block. The comments specify which conditions are true or false in each block.

```
// Change the values of these variables to test the results.  
bool Condition1 = true;  
bool Condition2 = true;  
bool Condition3 = true;  
bool Condition4 = true;  
  
if (Condition1)  
{  
    // Condition1 is true.  
}  
else if (Condition2)  
{  
    // Condition1 is false and Condition2 is true.  
}  
else if (Condition3)  
{  
    if (Condition4)  
    {  
        // Condition1 and Condition2 are false. Condition3 and Condition4 are true.  
    }  
    else  
    {  
        // Condition1, Condition2, and Condition4 are false. Condition3 is true.  
    }  
}  
else  
{  
    // Condition1, Condition2, and Condition3 are false.  
}
```

Example

The following example determines whether an input character is a lowercase letter, an uppercase letter, or a number. If all three conditions are false, the character isn't an alphanumeric character. The example displays a message for each case.

```
Console.Write("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

Just as a statement in the else block or the then block can be any valid statement, you can use any valid Boolean expression for the condition. You can use [logical operators](#) such as `!`, `&&`, `||`, `&`, `|`, and `^` to make compound conditions. The following code shows examples.

```

// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [?: Operator](#)
- [if-else Statement \(C++\)](#)
- [switch](#)

switch (C# reference)

11/2/2020 • 16 minutes to read • [Edit Online](#)

This article covers the `switch` statement. For information on the `switch` expression (introduced in C# 8.0), see the article on [switch expressions](#) in the [expressions and operators](#) section.

`switch` is a selection statement that chooses a single *switch section* to execute from a list of candidates based on a pattern match with the *match expression*.

```
using System;

public class Example
{
    public static void Main()
    {
        int caseSwitch = 1;

        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
                Console.WriteLine("Case 2");
                break;
            default:
                Console.WriteLine("Default case");
                break;
        }
    }
}

// The example displays the following output:
//      Case 1
```

The `switch` statement is often used as an alternative to an `if-else` construct if a single expression is tested against three or more conditions. For example, the following `switch` statement determines whether a variable of type `Color` has one of three values:

```

using System;

public enum Color { Red, Green, Blue }

public class Example
{
    public static void Main()
    {
        Color c = (Color) (new Random()).Next(0, 3);
        switch (c)
        {
            case Color.Red:
                Console.WriteLine("The color is red");
                break;
            case Color.Green:
                Console.WriteLine("The color is green");
                break;
            case Color.Blue:
                Console.WriteLine("The color is blue");
                break;
            default:
                Console.WriteLine("The color is unknown.");
                break;
        }
    }
}

```

It's equivalent to the following example that uses an `if`-`else` construct.

```

using System;

public enum Color { Red, Green, Blue }

public class Example
{
    public static void Main()
    {
        Color c = (Color) (new Random()).Next(0, 3);
        if (c == Color.Red)
            Console.WriteLine("The color is red");
        else if (c == Color.Green)
            Console.WriteLine("The color is green");
        else if (c == Color.Blue)
            Console.WriteLine("The color is blue");
        else
            Console.WriteLine("The color is unknown.");
    }
}

// The example displays the following output:
//      The color is red

```

The match expression

The match expression provides the value to match against the patterns in `case` labels. Its syntax is:

```
switch (expr)
```

In C# 6 and earlier, the match expression must be an expression that returns a value of the following types:

- a `char`.
- a `string`.

- a `bool`.
- an `integral` value, such as an `int` or a `long`.
- an `enum` value.

Starting with C# 7.0, the match expression can be any non-null expression.

The switch section

A `switch` statement includes one or more switch sections. Each switch section contains one or more *case labels* (either a case or default label) followed by one or more statements. The `switch` statement may include at most one default label placed in any switch section. The following example shows a simple `switch` statement that has three switch sections, each containing two statements. The second switch section contains the `case 2:` and `case 3:` labels.

A `switch` statement can include any number of switch sections, and each section can have one or more case labels, as shown in the following example. However, no two case labels may contain the same expression.

```
using System;

public class Example
{
    public static void Main()
    {
        Random rnd = new Random();
        int caseSwitch = rnd.Next(1,4);

        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
            case 3:
                Console.WriteLine($"Case {caseSwitch}");
                break;
            default:
                Console.WriteLine($"An unexpected value ({caseSwitch})");
                break;
        }
    }
}

// The example displays output like the following:
//      Case 1
```

Only one switch section in a switch statement executes. C# doesn't allow execution to continue from one switch section to the next. Because of this, the following code generates a compiler error, CS0163: "Control cannot fall through from one case label (<case label>) to another."

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1... ");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

This requirement is usually met by explicitly exiting the switch section by using a `break`, `goto`, or `return`

statement. However, the following code is also valid, because it ensures that program control can't fall through to the `default` switch section.

```
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1...");
        break;
    case 2:
    case 3:
        Console.WriteLine("... and/or Case 2");
        break;
    case 4:
        while (true)
            Console.WriteLine("Endless looping. . . .");
    default:
        Console.WriteLine("Default value...");
        break;
}
```

Execution of the statement list in the switch section with a case label that matches the match expression begins with the first statement and proceeds through the statement list, typically until a jump statement, such as a `break`, `goto case`, `goto label`, `return`, or `throw`, is reached. At that point, control is transferred outside the `switch` statement or to another case label. A `goto` statement, if it's used, must transfer control to a constant label. This restriction is necessary, since attempting to transfer control to a non-constant label can have undesirable side-effects, such as transferring control to an unintended location in code or creating an endless loop.

Case labels

Each case label specifies a pattern to compare to the match expression (the `caseSwitch` variable in the previous examples). If they match, control is transferred to the switch section that contains the **first** matching case label. If no case label pattern matches the match expression, control is transferred to the section with the `default` case label, if there's one. If there's no `default` case, no statements in any switch section are executed, and control is transferred outside the `switch` statement.

For information on the `switch` statement and pattern matching, see the [Pattern matching with the `switch` statement](#) section.

Because C# 6 supports only the constant pattern and doesn't allow the repetition of constant values, case labels define mutually exclusive values, and only one pattern can match the match expression. As a result, the order in which `case` statements appear is unimportant.

In C# 7.0, however, because other patterns are supported, case labels need not define mutually exclusive values, and multiple patterns can match the match expression. Because only the statements in the first switch section that contains the matching pattern are executed, the order in which `case` statements appear is now important. If C# detects a switch section whose case statement or statements are equivalent to or are subsets of previous statements, it generates a compiler error, CS8120, "The switch case has already been handled by a previous case."

The following example illustrates a `switch` statement that uses a variety of non-mutually exclusive patterns. If you move the `case 0:` switch section so that it's no longer the first section in the `switch` statement, C# generates a compiler error because an integer whose value is zero is a subset of all integers, which is the pattern defined by the `case int val` statement.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
public class Example
{
    public static void Main()
    {
        var values = new List<object>();
        for (int ctr = 0; ctr <= 7; ctr++) {
            if (ctr == 2)
                values.Add(DiceLibrary.Roll2());
            else if (ctr == 4)
                values.Add(DiceLibrary.Pass());
            else
                values.Add(DiceLibrary.Roll());
        }

        Console.WriteLine($"The sum of { values.Count } die is { DiceLibrary.DiceSum(values) }");
    }
}

public static class DiceLibrary
{
    // Random number generator to simulate dice rolls.
    static Random rnd = new Random();

    // Roll a single die.
    public static int Roll()
    {
        return rnd.Next(1, 7);
    }

    // Roll two dice.
    public static List<object> Roll2()
    {
        var rolls = new List<object>();
        rolls.Add(Roll());
        rolls.Add(Roll());
        return rolls;
    }

    // Calculate the sum of n dice rolls.
    public static int DiceSum(IEnumerable<object> values)
    {
        var sum = 0;
        foreach (var item in values)
        {
            switch (item)
            {
                // A single zero value.
                case 0:
                    break;
                // A single value.
                case int val:
                    sum += val;
                    break;
                // A non-empty collection.
                case IEnumerable<object> subList when subList.Any():
                    sum += DiceSum(subList);
                    break;
                // An empty collection.
                case IEnumerable<object> subList:
                    break;
                // A null reference.
                case null:
                    break;
                // A value that is neither an integer nor a collection.
                default:
                    throw new InvalidOperationException("unknown item type");
            }
        }
    }
}
```

```

        return sum;
    }

    public static object Pass()
    {
        if (rnd.Next(0, 2) == 0)
            return null;
        else
            return new List<object>();
    }
}

```

You can correct this issue and eliminate the compiler warning in one of two ways:

- By changing the order of the switch sections.
- By using a [when clause](#) in the `case` label.

The `default` case

The `default` case specifies the switch section to execute if the match expression doesn't match any other `case` label. If a `default` case is not present and the match expression doesn't match any other `case` label, program flow falls through the `switch` statement.

The `default` case can appear in any order in the `switch` statement. Regardless of its order in the source code, it's always evaluated last, after all `case` labels have been evaluated.

Pattern matching with the `switch` statement

Each `case` statement defines a pattern that, if it matches the match expression, causes its containing switch section to be executed. All versions of C# support the constant pattern. The remaining patterns are supported beginning with C# 7.0.

Constant pattern

The constant pattern tests whether the match expression equals a specified constant. Its syntax is:

```
case constant:
```

where *constant* is the value to test for. *constant* can be any of the following constant expressions:

- A `bool` literal: either `true` or `false`.
- Any `integral` constant, such as an `int`, a `long`, or a `byte`.
- The name of a declared `const` variable.
- An enumeration constant.
- A `char` literal.
- A `string` literal.

The constant expression is evaluated as follows:

- If *expr* and *constant* are integral types, the C# equality operator determines whether the expression returns `true` (that is, whether `expr == constant`).
- Otherwise, the value of the expression is determined by a call to the static `Object.Equals(expr, constant)` method.

The following example uses the constant pattern to determine whether a particular date is a weekend, the first day of the work week, the last day of the work week, or the middle of the work week. It evaluates the

`DateTime.DayOfWeek` property of the current day against the members of the `DayOfWeek` enumeration.

```
using System;

class Program
{
    static void Main()
    {
        switch (DateTime.Now.DayOfWeek)
        {
            case DayOfWeek.Sunday:
            case DayOfWeek.Saturday:
                Console.WriteLine("The weekend");
                break;
            case DayOfWeek.Monday:
                Console.WriteLine("The first day of the work week.");
                break;
            case DayOfWeek.Friday:
                Console.WriteLine("The last day of the work week.");
                break;
            default:
                Console.WriteLine("The middle of the work week.");
                break;
        }
    }
}

// The example displays output like the following:
//      The middle of the work week.
```

The following example uses the constant pattern to handle user input in a console application that simulates an automatic coffee machine.

```

using System;

class Example
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large");
        Console.Write("Please enter your selection: ");
        string str = Console.ReadLine();
        int cost = 0;

        // Because of the goto statements in cases 2 and 3, the base cost of 25
        // cents is added to the additional cost for the medium and large sizes.
        switch (str)
        {
            case "1":
            case "small":
                cost += 25;
                break;
            case "2":
            case "medium":
                cost += 25;
                goto case "1";
            case "3":
            case "large":
                cost += 50;
                goto case "1";
            default:
                Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Please insert {0} cents.", cost);
        }
        Console.WriteLine("Thank you for your business.");
    }
}

// The example displays output like the following:
//      Coffee sizes: 1=small 2=medium 3=large
//      Please enter your selection: 2
//      Please insert 50 cents.
//      Thank you for your business.

```

Type pattern

The type pattern enables concise type evaluation and conversion. When used with the `switch` statement to perform pattern matching, it tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type. Its syntax is:

```
case type varname
```

where *type* is the name of the type to which the result of *expr* is to be converted, and *varname* is the object to which the result of *expr* is converted if the match succeeds. The compile-time type of *expr* may be a generic type parameter, starting with C# 7.1.

The `case` expression is `true` if any of the following is true:

- *expr* is an instance of the same type as *type*.
- *expr* is an instance of a type that derives from *type*. In other words, the result of *expr* can be upcast to an instance of *type*.

- *expr* has a compile-time type that is a base class of *type*, and *expr* has a runtime type that is *type* or is derived from *type*. The *compile-time type* of a variable is the variable's type as defined in its type declaration. The *runtime type* of a variable is the type of the instance that is assigned to that variable.
- *expr* is an instance of a type that implements the *type* interface.

If the case expression is true, *varname* is definitely assigned and has local scope within the switch section only.

Note that `null` doesn't match a type. To match a `null`, you use the following `case` label:

```
case null:
```

The following example uses the type pattern to provide information about various kinds of collection types.

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        switch (coll)
        {
            case Array arr:
                Console.WriteLine($"An array with {arr.Length} elements.");
                break;
            case IEnumerable<int> ieInt:
                Console.WriteLine($"Average: {ieInt.Average(s => s)}");
                break;
            case IList list:
                Console.WriteLine($"{list.Count} items");
                break;
            case IEnumerable ie:
                string result = "";
                foreach (var e in ie)
                    result += $"{e} ";
                Console.WriteLine(result);
                break;
            case null:
                // Do nothing for a null.
                break;
            default:
                Console.WriteLine($"A instance of type {coll.GetType().Name}");
                break;
        }
    }
}

// The example displays the following output:
//      An array with 5 elements.
//      4 items

```

Instead of `object`, you could make a generic method, using the type of the collection as the type parameter, as shown in the following code:

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation<T>(T coll)
    {
        switch (coll)
        {
            case Array arr:
                Console.WriteLine($"An array with {arr.Length} elements.");
                break;
            case IEnumerable<int> ieInt:
                Console.WriteLine($"Average: {ieInt.Average(s => s)}");
                break;
            case IList list:
                Console.WriteLine($"{list.Count} items");
                break;
            case IEnumerable ie:
                string result = "";
                foreach (var e in ie)
                    result += $"{e} ";
                Console.WriteLine(result);
                break;
            case object o:
                Console.WriteLine($"A instance of type {o.GetType().Name}");
                break;
            default:
                Console.WriteLine("Null passed to this method.");
                break;
        }
    }
}

// The example displays the following output:
//     An array with 5 elements.
//     4 items
//     Null passed to this method.

```

The generic version is different than the first sample in two ways. First, you can't use the `null` case. You can't use any constant case because the compiler can't convert any arbitrary type `T` to any type other than `object`. What had been the `default` case now tests for a non-null `object`. That means the `default` case tests only for `null`.

Without pattern matching, this code might be written as follows. The use of type pattern matching produces more compact, readable code by eliminating the need to test whether the result of a conversion is a `null` or to perform repeated casts.

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        if (coll is Array)
        {
            Array arr = (Array) coll;
            Console.WriteLine($"An array with {arr.Length} elements.");
        }
        else if (coll is IEnumerable<int>)
        {
            IEnumerable<int> ieInt = (IEnumerable<int>) coll;
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
        }
        else if (coll is IList)
        {
            IList list = (IList) coll;
            Console.WriteLine($"{list.Count} items");
        }
        else if (coll is IEnumerable)
        {
            IEnumerable ie = (IEnumerable) coll;
            string result = "";
            foreach (var e in ie)
                result += $"{e} ";
            Console.WriteLine(result);
        }
        else if (coll == null)
        {
            // Do nothing.
        }
        else
        {
            Console.WriteLine($"An instance of type {coll.GetType().Name}");
        }
    }
}

// The example displays the following output:
//     An array with 5 elements.
//     4 items

```

The `case` statement and the `when` clause

Starting with C# 7.0, because case statements need not be mutually exclusive, you can add a `when` clause to specify an additional condition that must be satisfied for the case statement to evaluate to true. The `when` clause can be any expression that returns a Boolean value.

The following example defines a base `Shape` class, a `Rectangle` class that derives from `Shape`, and a `Square` class that derives from `Rectangle`. It uses the `when` clause to ensure that the `ShowShapeInfo` treats a `Rectangle` object that has been assigned equal lengths and widths as a `Square` even if it hasn't been instantiated as a `Square` object. The method doesn't attempt to display information either about an object that is `null` or a shape whose area is zero.

```
using System;

public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; set; }
    public double Width { get; set; }

    public override double Area
    {
        get { return Math.Round(Length * Width,2); }
    }

    public override double Circumference
    {
        get { return (Length + Width) * 2; }
    }
}

public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
        Side = side;
    }

    public double Side { get; set; }
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public override double Circumference
    {
        get { return 2 * Math.PI * Radius; }
    }

    public override double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

```

public class Example
{
    public static void Main()
    {
        Shape sh = null;
        Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                           sh, new Square(0), new Rectangle(8, 8),
                           new Circle(3) };
        foreach (var shape in shapes)
            ShowShapeInfo(shape);
    }

    private static void ShowShapeInfo(Shape sh)
    {
        switch (sh)
        {
            // Note that this code never evaluates to true.
            case Shape shape when shape == null:
                Console.WriteLine($"An uninitialized shape (shape == null)");
                break;
            case null:
                Console.WriteLine($"An uninitialized shape");
                break;
            case Shape shape when sh.Area == 0:
                Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
                break;
            case Square sq when sh.Area > 0:
                Console.WriteLine("Information about square:");
                Console.WriteLine($"    Length of a side: {sq.Side}");
                Console.WriteLine($"    Area: {sq.Area}");
                break;
            case Rectangle r when r.Length == r.Width && r.Area > 0:
                Console.WriteLine("Information about square rectangle:");
                Console.WriteLine($"    Length of a side: {r.Length}");
                Console.WriteLine($"    Area: {r.Area}");
                break;
            case Rectangle r when sh.Area > 0:
                Console.WriteLine("Information about rectangle:");
                Console.WriteLine($"    Dimensions: {r.Length} x {r.Width}");
                Console.WriteLine($"    Area: {r.Area}");
                break;
            case Shape shape when sh != null:
                Console.WriteLine($"A {sh.GetType().Name} shape");
                break;
            default:
                Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
                break;
        }
    }
}

// The example displays the following output:
//      Information about square:
//          Length of a side: 10
//          Area: 100
//      Information about rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//      An uninitialized shape
//      The shape: Square with no dimensions
//      Information about square rectangle:
//          Length of a side: 8
//          Area: 64
//      A Circle shape

```

Note that the `when` clause in the example that attempts to test whether a `Shape` object is `null` doesn't execute. The correct type pattern to test for a `null` is `case null:`.

C# language specification

For more information, see [The switch statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [if-else](#)
- [Pattern Matching](#)

do (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `do` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. Because that expression is evaluated after each execution of the loop, a `do-while` loop executes one or more times. This differs from the `while` loop, which executes zero or more times.

At any point within the `do` statement block, you can break out of the loop by using the `break` statement.

You can step directly to the evaluation of the `while` expression by using the `continue` statement. If the expression evaluates to `true`, execution continues at the first statement in the loop. Otherwise, execution continues at the first statement after the loop.

You can also exit a `do-while` loop by the `goto`, `return`, or `throw` statements.

Example

The following example shows the usage of the `do` statement. Select **Run** to run the example code. After that you can modify the code and run it again.

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

C# language specification

For more information, see [The do statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [while statement](#)

for (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `for` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`.

At any point within the `for` statement block, you can break out of the loop by using the `break` statement, or step to the next iteration in the loop by using the `continue` statement. You can also exit a `for` loop by the `goto`, `return`, or `throw` statements.

Structure of the `for` statement

The `for` statement defines *initializer*, *condition*, and *iterator* sections:

```
for (initializer; condition; iterator)
    body
```

All three sections are optional. The body of the loop is either a statement or a block of statements.

The following example shows the `for` statement with all of the sections defined:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

The *initializer* section

The statements in the *initializer* section are executed only once, before entering the loop. The *initializer* section is either of the following:

- The declaration and initialization of a local loop variable, which can't be accessed from outside the loop.
- Zero or more statement expressions from the following list, separated by commas:
 - `assignment` statement
 - invocation of a method
 - prefix or postfix `increment` expression, such as `++i` or `i++`
 - prefix or postfix `decrement` expression, such as `--i` or `i--`
 - creation of an object by using the `new` operator
 - `await` expression

The *initializer* section in the example above declares and initializes the local loop variable `i`:

```
int i = 0
```

The *condition* section

The *condition* section, if present, must be a boolean expression. That expression is evaluated before every loop

iteration. If the *condition* section is not present or the boolean expression evaluates to `true`, the next loop iteration is executed; otherwise, the loop is exited.

The *condition* section in the example above determines if the loop terminates based on the value of the local loop variable:

```
i < 5
```

The *iterator* section

The *iterator* section defines what happens after each iteration of the body of the loop. The *iterator* section contains zero or more of the following statement expressions, separated by commas:

- [assignment](#) statement
- invocation of a method
- prefix or postfix [increment](#) expression, such as `++i` or `i++`
- prefix or postfix [decrement](#) expression, such as `--i` or `i--`
- creation of an object by using the [new](#) operator
- [await](#) expression

The *iterator* section in the example above increments the local loop variable:

```
i++
```

Examples

The following example illustrates several less common usages of the `for` statement sections: assigning a value to an external loop variable in the *initializer* section, invoking a method in both the *initializer* and the *iterator* sections, and changing the values of two variables in the *iterator* section. Select **Run** to run the example code. After that you can modify the code and run it again.

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i={i}, j={j}"))
{
    // Body of the loop.
}
```

The following example defines the infinite `for` loop:

```
for ( ; ; )
{
    // Body of the loop.
}
```

C# language specification

For more information, see [The for statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [foreach, in](#)

foreach, in (C# reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `foreach` statement executes a statement or a block of statements for each element in an instance of the type that implements the `System.Collections.IEnumerable` or `System.Collections.Generic.IEnumerable<T>` interface, as the following example shows:

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
    Console.WriteLine($"Element #{count}: {element}");
    count++;
}
Console.WriteLine($"Number of elements: {count}");
```

The `foreach` statement isn't limited to those types. You can use it with an instance of any type that satisfies the following conditions:

- A type has the public parameterless `GetEnumerator` method whose return type is either class, struct, or interface type. Beginning with C# 9.0, the `GetEnumerator` method can be a type's [extension method](#).
- The return type of the `GetEnumerator` method has the public `current` property and the public parameterless `MoveNext` method whose return type is `Boolean`.

The following example uses the `foreach` statement with an instance of the `System.Span<T>` type, which doesn't implement any interfaces:

```
public class IterateSpanExample
{
    public static void Main()
    {
        Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
        foreach (int number in numbers)
        {
            Console.Write($"{number} ");
        }
        Console.WriteLine();
    }
}
```

Beginning with C# 7.3, if the enumerator's `Current` property returns a [reference return value](#) (`ref T` where `T` is the type of a collection element), you can declare an iteration variable with the `ref` or `ref readonly` modifier, as the following example shows:

```

public class ForeachRefExample
{
    public static void Main()
    {
        Span<int> storage = stackalloc int[10];
        int num = 0;
        foreach (ref int item in storage)
        {
            item = num++;
        }

        foreach (ref readonly var item in storage)
        {
            Console.WriteLine($"{item} ");
        }
        // Output:
        // 0 1 2 3 4 5 6 7 8 9
    }
}

```

Beginning with C# 8.0, you can use the `await foreach` statement to consume an asynchronous stream of data, that is, the collection type that implements the `IAsyncEnumerable<T>` interface. Each iteration of the loop may be suspended while the next element is retrieved asynchronously. The following example shows how to use the `await foreach` statement:

```

await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}

```

By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the `TaskAsyncEnumerableExtensions.ConfigureAwait` extension method. For more information about synchronization contexts and capturing the current context, see [Consuming the Task-based asynchronous pattern](#). For more information about asynchronous streams, see the [Asynchronous streams](#) section of the [What's new in C# 8.0](#) article.

At any point within the `foreach` statement block, you can break out of the loop by using the `break` statement, or step to the next iteration in the loop by using the `continue` statement. You can also exit a `foreach` loop by the `goto`, `return`, or `throw` statements.

If the `foreach` statement is applied to `null`, a `NullReferenceException` is thrown. If the source collection of the `foreach` statement is empty, the body of the `foreach` loop isn't executed and skipped.

Type of an iteration variable

You can use the `var` keyword to let the compiler infer the type of an iteration variable in the `foreach` statement, as the following code shows:

```

foreach (var item in collection) { }

```

You can also explicitly specify the type of an iteration variable, as the following code shows:

```

IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }

```

In the preceding form, type `T` of a collection element must be implicitly or explicitly convertible to type `V` of

an iteration variable. If an explicit conversion from `T` to `V` fails at run time, the `foreach` statement throws an [InvalidCastException](#). For example, if `T` is a non-sealed class type, `V` can be any interface type, even the one that `T` doesn't implement. At run time, the type of a collection element may be the one that derives from `T` and actually implements `V`. If that's not the case, an [InvalidCastException](#) is thrown.

C# language specification

For more information, see [The foreach statement](#) section of the [C# language specification](#).

For more information about features added in C# 8.0 and later, see the following feature proposal notes:

- [Async streams \(C# 8.0\)](#)
- [Extension `GetEnumerator` support for `foreach` loops \(C# 9.0\)](#)

See also

- [C# reference](#)
- [C# keywords](#)
- [Using foreach with arrays](#)
- [for statement](#)

while (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `while` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. Because that expression is evaluated before each execution of the loop, a `while` loop executes zero or more times. This differs from the `do` loop, which executes one or more times.

At any point within the `while` statement block, you can break out of the loop by using the `break` statement.

You can step directly to the evaluation of the `while` expression by using the `continue` statement. If the expression evaluates to `true`, execution continues at the first statement in the loop. Otherwise, execution continues at the first statement after the loop.

You can also exit a `while` loop by the `goto`, `return`, or `throw` statements.

Example

The following example shows the usage of the `while` statement. Select **Run** to run the example code. After that you can modify the code and run it again.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

C# language specification

For more information, see [The while statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [do statement](#)

break (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `break` statement terminates the closest enclosing loop or `switch` statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.

Example

In this example, the conditional statement contains a counter that is supposed to count from 1 to 100; however, the `break` statement terminates the loop after 4 counts.

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Output:
 1
 2
 3
 4
*/
```

Example

This example demonstrates the use of `break` in a `switch` statement.

```

class Switch
{
    static void Main()
    {
        Console.WriteLine("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is 1");
                break;
            case 2:
                Console.WriteLine("Current value is 2");
                break;
            case 3:
                Console.WriteLine("Current value is 3");
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/

```

If you entered **4**, the output would be:

```

Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.

```

Example

In this example, the **break** statement is used to break out of an inner nested loop, and return control to the outer loop. Control is *only* returned one level up in the nested loops.

```

class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

        // Outer loop.
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine($"num = {numbers[i]}");

            // Inner loop.
            for (int j = 0; j < letters.Length; j++)
            {
                if (j == i)
                {
                    // Return control to outer loop.
                    break;
                }
                Console.Write($"{letters[j]} ");
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
 num = 0

 num = 1
 a
 num = 2
 a b
 num = 3
 a b c
 num = 4
 a b c d
 num = 5
 a b c d e
 num = 6
 a b c d e f
 num = 7
 a b c d e f g
 num = 8
 a b c d e f g h
 num = 9
 a b c d e f g h i
*/

```

Example

In this example, the `break` statement is only used to break out of the current branch during each iteration of the loop. The loop itself is unaffected by the instances of `break` that belong to the nested `switch` statement.

```
class BreakFromSwitchInsideLoop
{
    static void Main(string[] args)
    {
        // loop 1 to 3
        for (int i = 1; i <= 3; i++)
        {
            switch(i)
            {
                case 1:
                    Console.WriteLine("Current value is 1");
                    break;
                case 2:
                    Console.WriteLine("Current value is 2");
                    break;
                case 3:
                    Console.WriteLine("Current value is 3");
                    break;
                default:
                    Console.WriteLine("This shouldn't happen.");
                    break;
            }
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
 * Current value is 1
 * Current value is 2
 * Current value is 3
*/
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [switch](#)

continue (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `continue` statement passes control to the next iteration of the enclosing `while`, `do`, `for`, or `foreach` statement in which it appears.

Example

In this example, a counter is initialized to count from 1 to 10. By using the `continue` statement in conjunction with the expression `(i < 9)`, the statements between `continue` and the end of the `for` body are skipped in the iterations where `i` is less than 9. In the last two iterations of the `for` loop (where `i == 9` and `i == 10`), the `continue` statement is not executed and the value of `i` is printed to the console.

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Output:
9
10
*/
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [break Statement](#)

goto (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `goto` statement transfers the program control directly to a labeled statement.

A common use of `goto` is to transfer control to a specific switch-case label or the default label in a `switch` statement.

The `goto` statement is also useful to get out of deeply nested loops.

Example

The following example demonstrates using `goto` in a `switch` statement.

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine($"Please insert {cost} cents.");
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Sample Input: 2

Sample Output:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
*/
```

Example

The following example demonstrates using `goto` to break out from nested loops.

```
public class GotoTest1
{
    static void Main()
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array.
        for (int i = 0; i < x; i++)
            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input.
        Console.Write("Enter the number to search for: ");

        // Input a string.
        string myNumber = Console.ReadLine();

        // Search.
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine($"The number {myNumber} was not found.");
        goto Finish;

    Found:
        Console.WriteLine($"The number {myNumber} is found.");

    Finish:
        Console.WriteLine("End of search.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [goto Statement \(C++\)](#)

return (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `return` statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value. If the method is a `void` type, the `return` statement can be omitted.

If the return statement is inside a `try` block, the `finally` block, if one exists, will be executed before control returns to the calling method.

Example

In the following example, the method `CalculateArea()` returns the local variable `area` as a `double` value.

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [return Statement](#)

throw (C# Reference)

4/3/2021 • 3 minutes to read • [Edit Online](#)

Signals the occurrence of an exception during program execution.

Remarks

The syntax of `throw` is:

```
throw [e];
```

where `e` is an instance of a class derived from [System.Exception](#). The following example uses the `throw` statement to throw an [IndexOutOfRangeException](#) if the argument passed to a method named `GetNumber` does not correspond to a valid index of an internal array.

```
using System;

namespace Throw2
{
    public class NumberGenerator
    {
        int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

        public int GetNumber(int index)
        {
            if (index < 0 || index >= numbers.Length)
            {
                throw new IndexOutOfRangeException();
            }
            return numbers[index];
        }
    }
}
```

Method callers then use a `try-catch` or `try-catch-finally` block to handle the thrown exception. The following example handles the exception thrown by the `GetNumber` method.

```
using System;

public class Example
{
    public static void Main()
    {
        var gen = new NumberGenerator();
        int index = 10;
        try
        {
            int value = gen.GetNumber(index);
            Console.WriteLine($"Retrieved {value}");
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
        }
    }
}

// The example displays the following output:
//     IndexOutOfRangeException: 10 is outside the bounds of the array
```

Re-throwing an exception

`throw` can also be used in a `catch` block to re-throw an exception handled in a `catch` block. In this case, `throw` does not take an exception operand. It is most useful when a method passes on an argument from a caller to some other library method, and the library method throws an exception that must be passed on to the caller. For example, the following example re-throws an [NullReferenceException](#) that is thrown when attempting to retrieve the first character of an uninitialized string.

```

using System;

namespace Throw
{
    public class Sentence
    {
        public Sentence(string s)
        {
            Value = s;
        }

        public string Value { get; set; }

        public char GetFirstCharacter()
        {
            try
            {
                return Value[0];
            }
            catch (NullReferenceException e)
            {
                throw;
            }
        }
    }

    public class Example
    {
        public static void Main()
        {
            var s = new Sentence(null);
            Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
        }
    }
}

// The example displays the following output:
//     Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//         at Sentence.GetFirstCharacter()
//         at Example.Main()

```

IMPORTANT

You can also use the `throw e` syntax in a `catch` block to instantiate a new exception that you pass on to the caller. In this case, the stack trace of the original exception, which is available from the `StackTrace` property, is not preserved.

The `throw` expression

Starting with C# 7.0, `throw` can be used as an expression as well as a statement. This allows an exception to be thrown in contexts that were previously unsupported. These include:

- **the conditional operator.** The following example uses a `throw` expression to throw an `ArgumentException` if a method is passed an empty string array. Before C# 7.0, this logic would need to appear in an `if / else` statement.

```
private static void DisplayFirstNumber(string[] args)
{
    string arg = args.Length >= 1 ? args[0] :
        throw new ArgumentException("You must supply an argument");
    if (Int64.TryParse(arg, out var number))
        Console.WriteLine($"You entered {number:F0}");
    else
        Console.WriteLine($"{arg} is not a number.");
}
```

- [the null-coalescing operator](#). In the following example, a `throw` expression is used with a null-coalescing operator to throw an exception if the string assigned to a `Name` property is `null`.

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

- an expression-bodied [lambda](#) or method. The following example illustrates an expression-bodied method that throws an [InvalidOperationException](#) because a conversion to a [DateTime](#) value is not supported.

```
DateTime ToDateTime(IFormatProvider provider) =>
    throw new InvalidOperationException("Conversion to a DateTime is not supported.");
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [try-catch](#)
- [C# Keywords](#)
- [How to: Explicitly Throw Exceptions](#)

try-catch (C# Reference)

11/2/2020 • 8 minutes to read • [Edit Online](#)

The try-catch statement consists of a `try` block followed by one or more `catch` clauses, which specify handlers for different exceptions.

When an exception is thrown, the common language runtime (CLR) looks for the `catch` statement that handles this exception. If the currently executing method does not contain such a `catch` block, the CLR looks at the method that called the current method, and so on up the call stack. If no `catch` block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

The `try` block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast a `null` object raises the [NullReferenceException](#) exception:

```
object o2 = null;
try
{
    int i2 = (int)o2;    // Error
}
```

Although the `catch` clause can be used without arguments to catch any type of exception, this usage is not recommended. In general, you should only catch those exceptions that you know how to recover from. Therefore, you should always specify an object argument derived from [System.Exception](#). For example:

```
catch (InvalidOperationException e)
{
}
```

It is possible to use more than one specific `catch` clause in the same try-catch statement. In this case, the order of the `catch` clauses is important because the `catch` clauses are examined in order. Catch the more specific exceptions before the less specific ones. The compiler produces an error if you order your catch blocks so that a later block can never be reached.

Using `catch` arguments is one way to filter for the exceptions you want to handle. You can also use an exception filter that further examines the exception to decide whether to handle it. If the exception filter returns false, then the search for a handler continues.

```
catch (ArgumentException e) when (e.ParamName == "...")
{
}
```

Exception filters are preferable to catching and rethrowing (explained below) because filters leave the stack unharmed. If a later handler dumps the stack, you can see where the exception originally came from, rather than just the last place it was rethrown. A common use of exception filter expressions is logging. You can create a filter that always returns false that also outputs to a log, you can log exceptions as they go by without having to handle them and rethrow.

A `throw` statement can be used in a `catch` block to re-throw the exception that is caught by the `catch` statement. The following example extracts source information from an [IOException](#) exception, and then throws

the exception to the parent method.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

You can catch one exception and throw a different exception. When you do this, specify the exception that you caught as the inner exception, as shown in the following example.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

You can also re-throw an exception when a specified condition is true, as shown in the following example.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

It is also possible to use an exception filter to get a similar result in an often cleaner fashion (as well as not modifying the stack, as explained earlier in this document). The following example has a similar behavior for callers as the previous example. The function throws the `InvalidCastException` back to the caller when `e.Data` is `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

From inside a `try` block, initialize only variables that are declared therein. Otherwise, an exception can occur before the execution of the block is completed. For example, in the following code example, the variable `n` is initialized inside the `try` block. An attempt to use this variable outside the `try` block in the `Write(n)` statement will generate a compiler error.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.WriteLine(n);
}
```

For more information about catch, see [try-catch-finally](#).

Exceptions in async methods

An async method is marked by an `async` modifier and usually contains one or more await expressions or statements. An await expression applies the `await` operator to a `Task` or `Task<TResult>`.

When control reaches an `await` in the async method, progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For more information, see [Asynchronous programming with async and await](#).

The completed task to which `await` is applied might be in a faulted state because of an unhandled exception in the method that returns the task. Awaiting the task throws an exception. A task can also end up in a canceled state if the asynchronous process that returns it is canceled. Awaiting a canceled task throws an `OperationCanceledException`.

To catch the exception, await the task in a `try` block, and catch the exception in the associated `catch` block. For an example, see the [Async method example](#) section.

A task can be in a faulted state because multiple exceptions occurred in the awaited async method. For example, the task might be the result of a call to `Task.WhenAll`. When you await such a task, only one of the exceptions is caught, and you can't predict which exception will be caught. For an example, see the [Task.WhenAll example](#) section.

Example

In the following example, the `try` block contains a call to the `ProcessString` method that may cause an exception. The `catch` clause contains the exception handler that just displays a message on the screen. When the `throw` statement is called from inside `ProcessString`, the system looks for the `catch` statement and displays the message `Exception caught`.

```

class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at TryFinallyTest.Main() Exception caught.
*/

```

Two catch blocks example

In the following example, two catch blocks are used, and the most specific exception, which comes first, is caught.

To catch the least specific exception, you can replace the throw statement in `ProcessString` with the following statement: `throw new Exception()`.

If you place the least-specific catch block first in the example, the following error message appears:

`A previous catch clause already catches all exceptions of this or a super type ('System.Exception').`

```

class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/

```

Async method example

The following example illustrates exception handling for async methods. To catch an exception that an async task throws, place the `await` expression in a `try` block, and catch the exception in a `catch` block.

Uncomment the `throw new Exception` line in the example to demonstrate exception handling. The task's `IsFaulted` property is set to `True`, the task's `Exception.InnerException` property is set to the exception, and the exception is caught in the `catch` block.

Uncomment the `throw new OperationCanceledException` line to demonstrate what happens when you cancel an asynchronous process. The task's `IsCanceled` property is set to `true`, and the exception is caught in the `catch` block. Under some conditions that don't apply to this example, the task's `IsFaulted` property is set to `true` and `IsCanceled` is set to `false`.

```

public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted: " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted: False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted: True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted: False

```

Task.WhenAll example

The following example illustrates exception handling where multiple tasks can result in multiple exceptions. The `try` block awaits the task that's returned by a call to [Task.WhenAll](#). The task is complete when the three tasks to which `WhenAll` is applied are complete.

Each of the three tasks causes an exception. The `catch` block iterates through the exceptions, which are found in the `Exception.InnerExceptions` property of the task that was returned by [Task.WhenAll](#).

```

public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerException)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task

```

C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [try-finally](#)
- [How to: Explicitly Throw Exceptions](#)

try-finally (C# Reference)

3/6/2021 • 3 minutes to read • [Edit Online](#)

By using a `finally` block, you can clean up any resources that are allocated in a `try` block, and you can run code even if an exception occurs in the `try` block. Typically, the statements of a `finally` block run when control leaves a `try` statement. The transfer of control can occur as a result of normal execution, of execution of a `break`, `continue`, `goto`, or `return` statement, or of propagation of an exception out of the `try` statement.

Within a handled exception, the associated `finally` block is guaranteed to be run. However, if the exception is unhandled, execution of the `finally` block is dependent on how the exception unwind operation is triggered. That, in turn, is dependent on how your computer is set up. The only cases where `finally` clauses don't run involve a program being immediately stopped. An example of this would be when `InvalidProgramException` gets thrown because of the IL statements being corrupt. On most operating systems, reasonable resource cleanup will take place as part of stopping and unloading the process.

Usually, when an unhandled exception ends an application, whether or not the `finally` block is run is not important. However, if you have statements in a `finally` block that must be run even in that situation, one solution is to add a `catch` block to the `try - finally` statement. Alternatively, you can catch the exception that might be thrown in the `try` block of a `try - finally` statement higher up the call stack. That is, you can catch the exception in the method that calls the method that contains the `try - finally` statement, or in the method that calls that method, or in any method in the call stack. If the exception is not caught, execution of the `finally` block depends on whether the operating system chooses to trigger an exception unwind operation.

Example

In the following example, an invalid conversion statement causes a `System.InvalidCastException` exception. The exception is unhandled.

```
public class ThrowTestA
{
    static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

In the following example, an exception from the `TryCast` method is caught in a method farther up the call stack.

```

public class ThrowTestB
{
    static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine(
                "Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    public static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
}

// Output:
// In the finally block in TryCast, i = 123.

// Catching the System.InvalidCastException exception triggers the finally block.

// Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}

```

For more information about `finally`, see [try-catch-finally](#).

C# also contains the [using statement](#), which provides similar functionality for [IDisposable](#) objects in a convenient syntax.

C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)

- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [try-catch](#)
- [How to: Explicitly Throw Exceptions](#)

try-catch-finally (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A common usage of `catch` and `finally` together is to obtain and use resources in a `try` block, deal with exceptional circumstances in a `catch` block, and release the resources in the `finally` block.

For more information and examples on re-throwing exceptions, see [try-catch](#) and [Throwing Exceptions](#). For more information about the `finally` block, see [try-finally](#).

Example

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [How to: Explicitly Throw Exceptions](#)
- [using Statement](#)

Checked and Unchecked (C# Reference)

3/15/2021 • 2 minutes to read • [Edit Online](#)

C# statements can execute in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated by discarding any high-order bits that don't fit in the destination type.

- [checked](#) Specify checked context.
- [unchecked](#) Specify unchecked context.

The following operations are affected by the overflow checking:

- Expressions using the following predefined operators on integral types:

`++`, `--`, unary `-`, `+`, `-`, `*`, `/`

- Explicit numeric conversions between integral types, or from `float` or `double` to an integral type.

If neither `checked` nor `unchecked` is specified, the default context for non-constant expressions (expressions that are evaluated at run time) is defined by the value of the [CheckForOverflowUnderflow](#) compiler option. By default the value of that option is unset and arithmetic operations are executed in an unchecked context.

For constant expressions (expressions that can be fully evaluated at compile time), the default context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression cause compile-time errors.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Statement Keywords](#)

checked (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `checked` keyword is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.

By default, an expression that contains only constant values causes a compiler error if the expression produces a value that is outside the range of the destination type. If the expression contains one or more non-constant values, the compiler does not detect the overflow. Evaluating the expression assigned to `i2` in the following example does not cause a compiler error.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

By default, these non-constant expressions are not checked for overflow at run time either, and they do not raise overflow exceptions. The previous example displays -2,147,483,639 as the sum of two positive integers.

Overflow checking can be enabled by compiler options, environment configuration, or use of the `checked` keyword. The following examples demonstrate how to use a `checked` expression or a `checked` block to detect the overflow that is produced by the previous sum at run time. Both examples raise an overflow exception.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

The `unchecked` keyword can be used to prevent overflow checking.

Example

This sample shows how to use `checked` to enable overflow checking at run time.

```

class OverFlowTest
{
    // Set maxValue to the maximum value for integers.
    static int maxValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT: " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT: " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
            CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
            UncheckedMethod());
    }
    /*
Output:
CHECKED and CAUGHT: System.OverflowException: Arithmetic operation resulted
in an overflow.
at ConsoleApplication1.OverFlowTest.CheckedMethod()

CHECKED output value is: 0
UNCHECKED output value is: -2147483639
*/
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Checked and Unchecked](#)
- [unchecked](#)

unchecked (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `unchecked` keyword is used to suppress overflow-checking for integral-type arithmetic operations and conversions.

In an unchecked context, if an expression produces a value that is outside the range of the destination type, the overflow is not flagged. For example, because the calculation in the following example is performed in an `unchecked` block or expression, the fact that the result is too large for an integer is ignored, and `int1` is assigned the value -2,147,483,639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

If the `unchecked` environment is removed, a compilation error occurs. The overflow can be detected at compile time because all the terms of the expression are constants.

Expressions that contain non-constant terms are unchecked by default at compile time and run time. See [checked](#) for information about enabling a checked environment.

Because checking for overflow takes time, the use of unchecked code in situations where there is no danger of overflow might improve performance. However, if overflow is a possibility, a checked environment should be used.

Example

This sample shows how to use the `unchecked` keyword.

```

class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for

C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Checked and Unchecked](#)
- [checked](#)

fixed Statement (C# Reference)

4/6/2021 • 3 minutes to read • [Edit Online](#)

The `fixed` statement prevents the garbage collector from relocating a movable variable. The `fixed` statement is only permitted in an `unsafe` context. You can also use the `fixed` keyword to create [fixed size buffers](#).

The `fixed` statement sets a pointer to a managed variable and "pins" that variable during the execution of the statement. Pointers to movable managed variables are useful only in a `fixed` context. Without a `fixed` context, garbage collection could relocate the variables unpredictably. The C# compiler only lets you assign a pointer to a managed variable in a `fixed` statement.

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

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

You can initialize a pointer by using an array, a string, a fixed-size buffer, or the address of a variable. The following example illustrates the use of variable addresses, arrays, and strings:

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

Starting with C# 7.3, the `fixed` statement operates on additional types beyond arrays, strings, fixed size buffers, or unmanaged variables. Any type that implements a method named `GetPinnableReference` can be pinned. The

`GetPinnableReference` must return a `ref` variable of an [unmanaged type](#). The .NET types `System.Span<T>` and `System.ReadOnlySpan<T>` introduced in .NET Core 2.0 make use of this pattern and can be pinned. This is shown in the following example:

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
        1,
        1, 1,
        1, 2, 1,
        1, 3, 3, 1,
        1, 4, 6, 4, 1,
        1, 5, 10, 10, 5, 1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

If you are creating types that should participate in this pattern, see [Span<T>.GetPinnableReference\(\)](#) for an example of implementing the pattern.

Multiple pointers can be initialized in one statement if they are all the same type:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

To initialize pointers of different types, simply nest `fixed` statements, as shown in the following example.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

After the code in the statement is executed, any pinned variables are unpinned and subject to garbage collection. Therefore, do not point to those variables outside the `fixed` statement. The variables declared in the `fixed` statement are scoped to that statement, making this easier:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    ...
}
// ps and pd are no longer in scope here.
```

Pointers initialized in `fixed` statements are readonly variables. If you want to modify the pointer value, you must declare a second pointer variable, and modify that. The variable declared in the `fixed` statement cannot be modified:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

You can allocate memory on the stack, where it is not subject to garbage collection and therefore does not need to be pinned. To do that, use a [stackalloc](#) expression.

C# language specification

For more information, see [The fixed statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [unsafe](#)
- [Pointer types](#)
- [Fixed Size Buffers](#)

lock statement (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `lock` statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.

The `lock` statement is of the form

```
lock (x)
{
    // Your code...
}
```

where `x` is an expression of a [reference type](#). It's precisely equivalent to

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Since the code uses a `try...finally` block, the lock is released even if an exception is thrown within the body of a `lock` statement.

You can't use the [await operator](#) in the body of a `lock` statement.

Guidelines

When you synchronize thread access to a shared resource, lock on a dedicated object instance (for example, `private readonly object balanceLock = new object();`) or another instance that is unlikely to be used as a lock object by unrelated parts of the code. Avoid using the same lock object instance for different shared resources, as it might result in deadlock or lock contention. In particular, avoid using the following as lock objects:

- `this`, as it might be used by the callers as a lock.
- `Type` instances, as those might be obtained by the `typeof` operator or reflection.
- `string` instances, including string literals, as those might be [interned](#).

Hold a lock for as short time as possible to reduce lock contention.

Example

The following example defines an `Account` class that synchronizes access to its private `balance` field by locking on a dedicated `balanceLock` instance. Using the same instance for locking ensures that the `balance` field cannot be updated simultaneously by two threads attempting to call the `debit` or `credit` methods simultaneously.

```

using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {

```

```
decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
foreach (var amount in amounts)
{
    if (amount >= 0)
    {
        account.Credit(amount);
    }
    else
    {
        account.Debit(Math.Abs(amount));
    }
}
```

C# language specification

For more information, see [The lock statement](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# keywords](#)
- [System.Threading.Monitor](#)
- [System.Threading.SpinLock](#)
- [System.Threading.Interlocked](#)
- [Overview of synchronization primitives](#)

Method Parameters (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Parameters declared for a method without `in`, `ref` or `out`, are passed to the called method by value. That value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure. By using a method parameter keyword, you can change this behavior.

This section describes the keywords you can use when declaring method parameters:

- `params` specifies that this parameter may take a variable number of arguments.
- `in` specifies that this parameter is passed by reference but is only read by the called method.
- `ref` specifies that this parameter is passed by reference and may be read or written by the called method.
- `out` specifies that this parameter is passed by reference and is written by the called method.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

params (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

By using the `params` keyword, you can specify a [method parameter](#) that takes a variable number of arguments. The parameter type must be a single-dimensional array.

No additional parameters are permitted after the `params` keyword in a method declaration, and only one `params` keyword is permitted in a method declaration.

If the declared type of the `params` parameter is not a single-dimensional array, compiler error [CS0225](#) occurs.

When you call a method with a `params` parameter, you can pass in:

- A comma-separated list of arguments of the type of the array elements.
- An array of arguments of the specified type.
- No arguments. If you send no arguments, the length of the `params` list is zero.

Example

The following example demonstrates various ways in which arguments can be sent to a `params` parameter.

```

public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}

/*
Output:
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Method Parameters](#)

in parameter modifier (C# Reference)

3/6/2021 • 5 minutes to read • [Edit Online](#)

The `in` keyword causes arguments to be passed by reference but ensures the argument is not modified. It makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument. It is like the `ref` or `out` keywords, except that `in` arguments cannot be modified by the called method. Whereas `ref` arguments may be modified, `out` arguments must be modified by the called method, and those modifications are observable in the calling context.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);      // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

The preceding example demonstrates that the `in` modifier is usually unnecessary at the call site. It is only required in the method declaration.

NOTE

The `in` keyword can also be used with a generic type parameter to specify that the type parameter is contravariant, as part of a `foreach` statement, or as part of a `join` clause in a LINQ query. For more information on the use of the `in` keyword in these contexts, see [in](#), which provides links to all those uses.

Variables passed as `in` arguments must be initialized before being passed in a method call. However, the called method may not assign a value or modify the argument.

The `in` parameter modifier is available in C# 7.2 and later. Previous versions generate compiler error `CS8107` ("Feature 'readonly references' is not available in C# 7.0. Please use language version 7.2 or greater.") To configure the compiler language version, see [Select the C# language version](#).

The `in`, `ref`, and `out` keywords are not considered part of the method signature for the purpose of overload resolution. Therefore, methods cannot be overloaded if the only difference is that one method takes a `ref` or `in` argument and the other takes an `out` argument. The following code, for example, will not compile:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Overloading based on the presence of `in` is allowed:

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Overload resolution rules

You can understand the overload resolution rules for methods with by value vs. `in` arguments by understanding the motivation for `in` arguments. Defining methods using `in` parameters is a potential performance optimization. Some `struct` type arguments may be large in size, and when methods are called in tight loops or critical code paths, the cost of copying those structures is critical. Methods declare `in` parameters to specify that arguments may be passed by reference safely because the called method does not modify the state of that argument. Passing those arguments by reference avoids the (potentially) expensive copy.

Specifying `in` for arguments at the call site is typically optional. There is no semantic difference between passing arguments by value and passing them by reference using the `in` modifier. The `in` modifier at the call site is optional because you don't need to indicate that the argument's value might be changed. You explicitly add the `in` modifier at the call site to ensure the argument is passed by reference, not by value. Explicitly using `in` has the following two effects:

First, specifying `in` at the call site forces the compiler to select a method defined with a matching `in` parameter. Otherwise, when two methods differ only in the presence of `in`, the by value overload is a better match.

Second, specifying `in` declares your intent to pass an argument by reference. The argument used with `in` must represent a location that can be directly referred to. The same general rules for `out` and `ref` arguments apply: You cannot use constants, ordinary properties, or other expressions that produce values. Otherwise, omitting `in` at the call site informs the compiler that you will allow it to create a temporary variable to pass by read-only reference to the method. The compiler creates a temporary variable to overcome several restrictions with `in` arguments:

- A temporary variable allows compile-time constants as `in` parameters.
- A temporary variable allows properties, or other expressions for `in` parameters.
- A temporary variable allows arguments where there is an implicit conversion from the argument type to the parameter type.

In all the preceding instances, the compiler creates a temporary variable that stores the value of the constant, property, or other expression.

The following code illustrates these rules:

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Now, suppose another method using by value arguments was available. The results change as shown in the following code:

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

The only method call where the argument is passed by reference is the final one.

NOTE

The preceding code uses `int` as the argument type for simplicity. Because `int` is no larger than a reference in most modern machines, there is no benefit to passing a single `int` as a readonly reference.

Limitations on `in` parameters

You can't use the `in`, `ref`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the `async` modifier.
- Iterator methods, which include a `yield return` or `yield break` statement.
- The first argument of an extension method cannot have the `in` modifier unless that argument is a struct.
- The first argument of an extension method where that argument is a generic type (even when that type is constrained to be a struct.)

You can learn more about the `in` modifier, how it differs from `ref` and `out` in the article on [Write safe efficient code](#).

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

ref (C# Reference)

3/31/2021 • 9 minutes to read • [Edit Online](#)

The `ref` keyword indicates that a value is passed by reference. It is used in four different contexts:

- In a method signature and in a method call, to pass an argument to a method by reference. For more information, see [Passing an argument by reference](#).
- In a method signature, to return a value to the caller by reference. For more information, see [Reference return values](#).
- In a member body, to indicate that a reference return value is stored locally as a reference that the caller intends to modify. Or to indicate that a local variable accesses another value by reference. For more information, see [Ref locals](#).
- In a `struct` declaration, to declare a `ref struct` or a `readonly ref struct`. For more information, see the `ref struct` section of the [Structure types](#) article.

Passing an argument by reference

When used in a method's parameter list, the `ref` keyword indicates that an argument is passed by reference, not by value. The `ref` keyword makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument.

For example, suppose the caller passes a local variable expression or an array element access expression. The called method can then replace the object to which the `ref` parameter refers. In that case, the caller's local variable or the array element refers to the new object when the method returns.

NOTE

Don't confuse the concept of passing by reference with the concept of reference types. The two concepts are not the same. A method parameter can be modified by `ref` regardless of whether it is a value type or a reference type. There is no boxing of a value type when it is passed by reference.

To use a `ref` parameter, both the method definition and the calling method must explicitly use the `ref` keyword, as shown in the following example. (Except that the calling method can omit `ref` when making a COM call.)

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

An argument that is passed to a `ref` or `in` parameter must be initialized before it is passed. This requirement differs from `out` parameters, whose arguments don't have to be explicitly initialized before they are passed.

Members of a class can't have signatures that differ only by `ref`, `in`, or `out`. A compiler error occurs if the only difference between two members of a type is that one of them has a `ref` parameter and the other has an `out`, or `in` parameter. The following code, for example, doesn't compile.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

However, methods can be overloaded when one method has a `ref`, `in`, or `out` parameter and the other has a parameter that is passed by value, as shown in the following example.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

In other situations that require signature matching, such as hiding or overriding, `in`, `ref`, and `out` are part of the signature and don't match each other.

Properties are not variables. They're methods, and cannot be passed to `ref` parameters.

You can't use the `ref`, `in`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the `async` modifier.
- Iterator methods, which include a `yield return` or `yield break` statement.

[extension methods](#) also have restrictions on the use of these keywords:

- The `out` keyword cannot be used on the first argument of an extension method.
- The `ref` keyword cannot be used on the first argument of an extension method when the argument is not a struct, or a generic type not constrained to be a struct.
- The `in` keyword cannot be used unless the first argument is a struct. The `in` keyword cannot be used on any generic type, even when constrained to be a struct.

Passing an argument by reference: An example

The previous examples pass value types by reference. You can also use the `ref` keyword to pass reference types by reference. Passing a reference type by reference enables the called method to replace the object to which the reference parameter refers in the caller. The storage location of the object is passed to the method as the value of the reference parameter. If you change the value in the storage location of the parameter (to point to a new object), you also change the storage location to which the caller refers. The following example passes an instance of a reference type as a `ref` parameter.

```

class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main. Name: Fasteners, ID: 54321
// Back in Main. Name: Stapler, ID: 12345

```

For more information about how to pass reference types by value and by reference, see [Passing Reference-Type Parameters](#).

Reference return values

Reference return values (or `ref` returns) are values that a method returns by reference to the caller. That is, the caller can modify the value returned by a method, and that change is reflected in the state of the object in the calling method.

A reference return value is defined by using the `ref` keyword:

- In the method signature. For example, the following method signature indicates that the `GetCurrentPrice` method returns a `Decimal` value by reference.

```
public ref decimal GetCurrentPrice()
```

- Between the `return` token and the variable returned in a `return` statement in the method. For example:

```
return ref DecimalArray[0];
```

In order for the caller to modify the object's state, the reference return value must be stored to a variable that is explicitly defined as a [ref local](#).

Here's a more complete ref return example, showing both the method signature and method body.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

The called method may also declare the return value as `ref readonly` to return the value by reference, and enforce that the calling code can't modify the returned value. The calling method can avoid copying the returned value by storing the value in a local `ref readonly` variable.

For an example, see [A ref returns and ref locals example](#).

Ref locals

A ref local variable is used to refer to values returned using `return ref`. A ref local variable cannot be initialized to a non-ref return value. In other words, the right-hand side of the initialization must be a reference. Any modifications to the value of the ref local are reflected in the state of the object whose method returned the value by reference.

You define a ref local by using the `ref` keyword in two places:

- Before the variable declaration.
- Immediately before the call to the method that returns the value by reference.

For example, the following statement defines a ref local value that is returned by a method named

`GetEstimatedValue` :

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how to define a ref local variable that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

In both examples the `ref` keyword must be used in both places, or the compiler generates error CS8172, "Cannot initialize a by-reference variable with a value."

Beginning with C# 7.3, the iteration variable of the `foreach` statement can be a ref local or ref readonly local variable. For more information, see the [foreach statement](#) article.

Also beginning with C# 7.3, you can reassign a ref local or ref readonly local variable with the [ref assignment operator](#).

Ref readonly locals

A ref readonly local is used to refer to values returned by a method or property that has `ref readonly` in its

signature and uses `return ref`. A `ref readonly` variable combines the properties of a `ref` local variable with a `readonly` variable: it's an alias to the storage it's assigned to, and it cannot be modified.

A `ref` returns and `ref locals` example

The following example defines a `Book` class that has two `String` fields, `Title` and `Author`. It also defines a `BookCollection` class that includes a private array of `Book` objects. Individual book objects are returned by reference by calling its `GetBookByTitle` method.

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                            new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" } };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

When the caller stores the value returned by the `GetBookByTitle` method as a `ref local`, changes that the caller makes to the return value are reflected in the `BookCollection` object, as the following example shows.

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//      Call of the Wild, The, by Jack London
//      Tale of Two Cities, A, by Charles Dickens
//
//      Republic, The, by Plato
//      Tale of Two Cities, A, by Charles Dickens
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Write safe efficient code](#)
- [Ref returns and ref locals](#)
- [Conditional ref expression](#)
- [Passing Parameters](#)
- [Method Parameters](#)
- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

out parameter modifier (C# Reference)

11/2/2020 • 4 minutes to read • [Edit Online](#)

The `out` keyword causes arguments to be passed by reference. It makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument. It is like the `ref` keyword, except that `ref` requires that the variable be initialized before it is passed. It is also like the `in` keyword, except that `in` does not allow the called method to modify the argument value. To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword. For example:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);      // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

NOTE

The `out` keyword can also be used with a generic type parameter to specify that the type parameter is covariant. For more information on the use of the `out` keyword in this context, see [out \(Generic Modifier\)](#).

Variables passed as `out` arguments do not have to be initialized before being passed in a method call. However, the called method is required to assign a value before the method returns.

The `in`, `ref`, and `out` keywords are not considered part of the method signature for the purpose of overload resolution. Therefore, methods cannot be overloaded if the only difference is that one method takes a `ref` or `in` argument and the other takes an `out` argument. The following code, for example, will not compile:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Overloading is legal, however, if one method takes a `ref`, `in`, or `out` argument and the other has none of those modifiers, like this:

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

The compiler chooses the best overload by matching the parameter modifiers at the call site to the parameter modifiers used in the method call.

Properties are not variables and therefore cannot be passed as `out` parameters.

You can't use the `in`, `ref`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the `async` modifier.
- Iterator methods, which include a `yield return` or `yield break` statement.

In addition, [extension methods](#) have the following restrictions:

- The `out` keyword cannot be used on the first argument of an extension method.
- The `ref` keyword cannot be used on the first argument of an extension method when the argument is not a struct, or a generic type not constrained to be a struct.
- The `in` keyword cannot be used unless the first argument is a struct. The `in` keyword cannot be used on any generic type, even when constrained to be a struct.

Declaring `out` parameters

Declaring a method with `out` arguments is a classic workaround to return multiple values. Beginning with C# 7.0, consider [value tuples](#) for similar scenarios. The following example uses `out` to return three variables with a single method call. The third argument is assigned to null. This enables methods to return values optionally.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

Calling a method with an `out` argument

In C# 6 and earlier, you must declare a variable in a separate statement before you pass it as an `out` argument. The following example declares a variable named `number` before it is passed to the `Int32.TryParse` method, which attempts to convert a string to a number.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

Starting with C# 7.0, you can declare the `out` variable in the argument list of the method call, rather than in a separate variable declaration. This produces more compact, readable code, and also prevents you from inadvertently assigning a value to the variable before the method call. The following example is like the previous example, except that it defines the `number` variable in the call to the `Int32.TryParse` method.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

In the previous example, the `number` variable is strongly typed as an `int`. You can also declare an implicitly typed local variable, as the following example does.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Method Parameters](#)

namespace (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `namespace` keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Remarks

Within a namespace, you can declare zero or more of the following types:

- another namespace
- [class](#)
- [interface](#)
- [struct](#)
- [enum](#)
- [delegate](#)

Whether or not you explicitly declare a namespace in a C# source file, the compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access and this is not modifiable. For a discussion of the access modifiers you can assign to elements in a namespace, see [Access Modifiers](#).

It is possible to define a namespace in two or more declarations. For example, the following example defines two classes as part of the `MyCompany` namespace:

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

Example

The following example shows how to call a static method in a nested namespace.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}

// Output: Hello
```

C# language specification

For more information, see the [Namespaces](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# keywords](#)
- [using](#)
- [using static](#)
- [Namespace alias qualifier `::`](#)
- [Namespaces](#)

using (C# Reference)

3/6/2021 • 2 minutes to read • [Edit Online](#)

The `using` keyword has three major uses:

- The [using statement](#) defines a scope at the end of which an object will be disposed.
- The [using directive](#) creates an alias for a namespace or imports types defined in other namespaces.
- The [using static directive](#) imports the members of a single class.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Namespaces](#)
- [extern](#)

using directive (C# Reference)

3/6/2021 • 3 minutes to read • [Edit Online](#)

The `using` directive has three uses:

- To allow the use of types in a namespace so that you do not have to qualify the use of a type in that namespace:

```
using System.Text;
```

- To allow you to access static members and nested types of a type without having to qualify the access with the type name.

```
using static System.Math;
```

For more information, see the [using static directive](#).

- To create an alias for a namespace or a type. This is called a *using alias directive*.

```
using Project = PC.MyCompany.Project;
```

The `using` keyword is also used to create *using statements*, which help ensure that [IDisposable](#) objects such as files and fonts are handled correctly. See [using Statement](#) for more information.

Using static type

You can access static members of a type without having to qualify the access with the type name:

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Remarks

The scope of a `using` directive is limited to the file in which it appears.

The `using` directive can appear:

- At the beginning of a source code file, before any namespace or type definitions.
- In any namespace, but before any namespace or types declared in this namespace.

Otherwise, compiler error [CS1529](#) is generated.

Create a `using` alias directive to make it easier to qualify an identifier to a namespace or type. In any `using` directive, the fully-qualified namespace or type must be used regardless of the `using` directives that come

before it. No `using` alias can be used in the declaration of a `using` directive. For example, the following generates a compiler error:

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

Create a `using` directive to use the types in a namespace without having to specify the namespace. A `using` directive does not give you access to any namespaces that are nested in the namespace you specify.

Namespaces come in two categories: user-defined and system-defined. User-defined namespaces are namespaces defined in your code. For a list of the system-defined namespaces, see [.NET API Browser](#).

Example 1

The following example shows how to define and use a `using` alias for a namespace:

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

A `using` alias directive cannot have an open generic type on the right hand side. For example, you cannot create a `using` alias for a `List<T>`, but you can create one for a `List<int>`.

Example 2

The following example shows how to define a `using` directive and a `using` alias for a class:

```

using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;
```

```

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//     You are in NameSpace1.MyClass.
//     You are in NameSpace2.MyClass.
```

C# language specification

For more information, see [Using directives](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Using Namespaces](#)
- [C# Keywords](#)
- [Namespaces](#)
- [using Statement](#)

using static directive (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `using static` directive designates a type whose static members and nested types you can access without specifying a type name. Its syntax is:

```
using static <fully-qualified-type-name>;
```

where *fully-qualified-type-name* is the name of the type whose static members and nested types can be referenced without specifying a type name. If you do not provide a fully qualified type name (the full namespace name along with the type name), C# generates compiler error [CS0246](#): "The type or namespace name 'type/namespace' could not be found (are you missing a using directive or an assembly reference?)".

The `using static` directive applies to any type that has static members (or nested types), even if it also has instance members. However, instance members can only be invoked through the type instance.

The `using static` directive was introduced in C# 6.

Remarks

Ordinarily, when you call a static member, you provide the type name along with the member name. Repeatedly entering the same type name to invoke members of the type can result in verbose, obscure code. For example, the following definition of a `Circle` class references a number of members of the `Math` class.

```
using System;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

By eliminating the need to explicitly reference the `Math` class each time a member is referenced, the `using static` directive produces much cleaner code:

```
using System;
using static System.Math;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
```

`using static` imports only accessible static members and nested types declared in the specified type. Inherited members are not imported. You can import from any named type with a `using static` directive, including Visual Basic modules. If F# top-level functions appear in metadata as static members of a named type whose name is a valid C# identifier, then the F# functions can be imported.

`using static` makes extension methods declared in the specified type available for extension method lookup. However, the names of the extension methods are not imported into scope for unqualified reference in code.

Methods with the same name imported from different types by different `using static` directives in the same compilation unit or namespace form a method group. Overload resolution within these method groups follows normal C# rules.

Example

The following example uses the `using static` directive to make the static members of the [Console](#), [Math](#), and [String](#) classes available without having to specify their type name.

```

using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
    static void Main()
    {
        Write("Enter a circle's radius: ");
        var input = ReadLine();
        if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
            var c = new Circle(radius);

            string s = "\nInformation about the circle:\n";
            s = s + Format("    Radius: {0:N2}\n", c.Radius);
            s = s + Format("    Diameter: {0:N2}\n", c.Diameter);
            s = s + Format("    Circumference: {0:N2}\n", c.Circumference);
            s = s + Format("    Area: {0:N2}\n", c.Area);
            WriteLine(s);
        }
        else {
            WriteLine("Invalid input...");
        }
    }
}

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
// The example displays the following output:
//      Enter a circle's radius: 12.45
//
//      Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95

```

In the example, the `using static` directive could also have been applied to the `Double` type. This would have made it possible to call the `TryParse(String, Double)` method without specifying a type name. However, this creates less readable code, since it becomes necessary to check the `using static` directives to determine which numeric type's `TryParse` method is called.

See also

- [using directive](#)
- [C# Reference](#)
- [C# Keywords](#)
- [Using Namespaces](#)
- [Namespaces](#)

using statement (C# Reference)

3/6/2021 • 4 minutes to read • [Edit Online](#)

Provides a convenient syntax that ensures the correct use of [IDisposable](#) objects. Beginning in C# 8.0, the `using` statement ensures the correct use of [IAsyncDisposable](#) objects.

Example

The following example shows how to use the `using` statement.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

Beginning with C# 8.0, you can use the following alternative syntax for the `using` statement that doesn't require braces:

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do {
    item = reader.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

Remarks

[File](#) and [Font](#) are examples of managed types that access unmanaged resources (in this case file handles and device contexts). There are many other kinds of unmanaged resources and class library types that encapsulate them. All such types must implement the [IDisposable](#) interface, or the [IAsyncDisposable](#) interface.

When the lifetime of an `IDisposable` object is limited to a single method, you should declare and instantiate it in the `using` statement. The `using` statement calls the [Dispose](#) method on the object in the correct way, and (when you use it as shown earlier) it also causes the object itself to go out of scope as soon as [Dispose](#) is called. Within the `using` block, the object is read-only and can't be modified or reassigned. If the object implements [IAsyncDisposable](#) instead of `IDisposable`, the `using` statement calls the [DisposeAsync](#) and [awaits](#) the returned [ValueTask](#). For more information on [IAsyncDisposable](#), see [Implement a DisposeAsync method](#).

The `using` statement ensures that `Dispose` (or `DisposeAsync`) is called even if an exception occurs within the `using` block. You can achieve the same result by putting the object inside a `try` block and then calling `Dispose` (or `DisposeAsync`) in a `finally` block; in fact, this is how the `using` statement is translated by the compiler. The code example earlier expands to the following code at compile time (note the extra curly braces to create the limited scope for the object):

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try {
        string? item;
        do {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while(item != null);
    } finally
    {
        reader?.Dispose();
    }
}
```

The newer `using` statement syntax translates to similar code. The `try` block opens where the variable is declared. The `finally` block is added at the close of the enclosing block, typically at the end of a method.

For more information about the `try - finally` statement, see the [try-finally](#) article.

Multiple instances of a type can be declared in a single `using` statement, as shown in the following example. Notice that you can't use implicitly typed variables (`var`) when you declare multiple variables in a single statement:

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
       right = new StringReader(letters))
{
    string? item;
    do {
        item = left.ReadLine();
        Console.Write(item);
        Console.Write("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

You can combine multiple declarations of the same type using the new syntax introduced with C# 8 as well, as shown in the following example:

```

string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using StringReader left = new StringReader(numbers),
      right = new StringReader(letters);
string? item;
do {
    item = left.ReadLine();
    Console.WriteLine(item);
    Console.WriteLine("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while(item != null);

```

You can instantiate the resource object and then pass the variable to the `using` statement, but this isn't a best practice. In this case, after control leaves the `using` block, the object remains in scope but probably has no access to its unmanaged resources. In other words, it's not fully initialized anymore. If you try to use the object outside the `using` block, you risk causing an exception to be thrown. For this reason, it's better to instantiate the object in the `using` statement and limit its scope to the `using` block.

```

string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
// reader is in scope here, but has been disposed

```

For more information about disposing of `IDisposable` objects, see [Using objects that implement IDisposable](#).

C# language specification

For more information, see [The using statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [using Directive](#)
- [Garbage Collection](#)
- [Using objects that implement IDisposable](#)

- [IDisposable interface](#)
- [using statement in C# 8.0](#)

extern alias (C# Reference)

3/15/2021 • 2 minutes to read • [Edit Online](#)

You might have to reference two versions of assemblies that have the same fully-qualified type names. For example, you might have to use two or more versions of an assembly in the same application. By using an external assembly alias, the namespaces from each assembly can be wrapped inside root-level namespaces named by the alias, which enables them to be used in the same file.

NOTE

The `extern` keyword is also used as a method modifier, declaring a method written in unmanaged code.

To reference two assemblies with the same fully-qualified type names, an alias must be specified at a command prompt, as follows:

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

This creates the external aliases `GridV1` and `GridV2`. To use these aliases from within a program, reference them by using the `extern` keyword. For example:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Each extern alias declaration introduces an additional root-level namespace that parallels (but does not lie within) the global namespace. Thus types from each assembly can be referred to without ambiguity by using their fully qualified name, rooted in the appropriate namespace-alias.

In the previous example, `GridV1::Grid` would be the grid control from `grid.dll`, and `GridV2::Grid` would be the grid control from `grid20.dll`.

Using Visual Studio

If you are using Visual Studio, aliases can be provided in similar way.

Add reference of `grid.dll` and `grid20.dll` to your project in Visual Studio. Open a property tab and change the Aliases from global to GridV1 and GridV2 respectively.

Use these aliases the same way above

```
extern alias GridV1;  
  
extern alias GridV2;
```

Now you can create alias for a namespace or a type by *using alias directive*. For more information, see [using directive](#).

```
using Class1V1 = GridV1::Namespace.Class1;  
  
using Class1V2 = GridV2::Namespace.Class1;
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [:: Operator](#)
- [References \(C# Compiler Options\)](#)

is (C# Reference)

3/6/2021 • 7 minutes to read • [Edit Online](#)

The `is` operator checks if the result of an expression is compatible with a given type, or (starting with C# 7.0) tests an expression against a pattern. For information about the type-testing `is` operator, see the [is operator](#) section of the [Type-testing and cast operators](#) article.

Pattern matching with `is`

Starting with C# 7.0, the `is` and `switch` statements support pattern matching. The `is` keyword supports the following patterns:

- [Type pattern](#), which tests whether an expression can be converted to a specified type and, if it can be, casts the variable to a variable of that type.
- [Constant pattern](#), which tests whether an expression evaluates to a specified constant value.
- [var pattern](#), a match that always succeeds and binds the value of an expression to a new local variable.

Type pattern

When using the type pattern to perform pattern matching, `is` tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type. It's a straightforward extension of the `is` statement that enables concise type evaluation and conversion. The general form of the `is` type pattern is:

```
expr is type varname
```

Where `expr` is an expression that evaluates to an instance of some type, `type` is the name of the type to which the result of `expr` is to be converted, and `varname` is the object to which the result of `expr` is converted if the `is` test is `true`.

The `is` expression is `true` if `expr` isn't `null`, and any of the following conditions is true:

- `expr` is an instance of the same type as `type`.
- `expr` is an instance of a type that derives from `type`. In other words, the result of `expr` can be upcast to an instance of `type`.
- `expr` has a compile-time type that is a base class of `type`, and `expr` has a runtime type that is `type` or is derived from `type`. The *compile-time type* of a variable is the variable's type as defined in its declaration. The *runtime type* of a variable is the type of the instance that is assigned to that variable.
- `expr` is an instance of a type that implements the `type` interface.

Beginning with C# 7.1, `expr` may have a compile-time type defined by a generic type parameter and its constraints.

If `expr` is `true` and `is` is used with an `if` statement, `varname` is assigned within the `if` statement only. The scope of `varname` is from the `is` expression to the end of the block enclosing the `if` statement. Using `varname` in any other location generates a compile-time error for use of a variable that hasn't been assigned.

The following example uses the `is` type pattern to provide the implementation of a type's `IComparable.CompareTo(Object)` method.

```

using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        if (o is Employee e)
        {
            return Name.CompareTo(e.Name);
        }
        throw new ArgumentException("o is not an Employee object.");
    }
}

```

Without pattern matching, this code might be written as follows. The use of type pattern matching produces more compact, readable code by eliminating the need to test whether the result of a conversion is a `null`.

```

using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        var e = o as Employee;
        if (e == null)
        {
            throw new ArgumentException("o is not an Employee object.");
        }
        return Name.CompareTo(e.Name);
    }
}

```

The `is` type pattern also produces more compact code when determining the type of a value type. The following example uses the `is` type pattern to determine whether an object is a `Person` or a `Dog` instance before displaying the value of an appropriate property.

```
using System;

public class Example
{
    public static void Main()
    {
        Object o = new Person("Jane");
        ShowValue(o);

        o = new Dog("Alaskan Malamute");
        ShowValue(o);
    }

    public static void ShowValue(object o)
    {
        if (o is Person p) {
            Console.WriteLine(p.Name);
        }
        else if (o is Dog d) {
            Console.WriteLine(d.Breed);
        }
    }
}

public struct Person
{
    public string Name { get; set; }

    public Person(string name) : this()
    {
        Name = name;
    }
}

public struct Dog
{
    public string Breed { get; set; }

    public Dog(string breedName) : this()
    {
        Breed = breedName;
    }
}
// The example displays the following output:
// Jane
// Alaskan Malamute
```

The equivalent code without pattern matching requires a separate assignment that includes an explicit cast.

```

using System;

public class Example
{
    public static void Main()
    {
        Object o = new Person("Jane");
        ShowValue(o);

        o = new Dog("Alaskan Malamute");
        ShowValue(o);
    }

    public static void ShowValue(object o)
    {
        if (o is Person) {
            Person p = (Person) o;
            Console.WriteLine(p.Name);
        }
        else if (o is Dog) {
            Dog d = (Dog) o;
            Console.WriteLine(d.Breed);
        }
    }
}

public struct Person
{
    public string Name { get; set; }

    public Person(string name) : this()
    {
        Name = name;
    }
}

public struct Dog
{
    public string Breed { get; set; }

    public Dog(string breedName) : this()
    {
        Breed = breedName;
    }
}
// The example displays the following output:
//      Jane
//      Alaskan Malamute

```

Constant pattern

When performing pattern matching with the constant pattern, `is` tests whether an expression equals a specified constant. In C# 6 and earlier versions, the constant pattern is supported by the `switch` statement. Starting with C# 7.0, it's supported by the `is` statement as well. Its syntax is:

```
expr is constant
```

where *expr* is the expression to evaluate, and *constant* is the value to test for. *constant* can be any of the following constant expressions:

- A literal value.
- The name of a declared `const` variable.

- An enumeration constant.

The constant expression is evaluated as follows:

- If *expr* and *constant* are integral types, the C# equality operator determines whether the expression returns `true` (that is, whether `expr == constant`).
- Otherwise, the value of the expression is determined by a call to the static `Object.Equals(expr, constant)` method.

The following example combines the type and constant patterns to test whether an object is a `Dice` instance and, if it is, to determine whether the value of a dice roll is 6.

```
using System;

public class Dice
{
    Random rnd = new Random();
    public Dice()
    {
    }
    public int Roll()
    {
        return rnd.Next(1, 7);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var d1 = new Dice();
        ShowValue(d1);
    }

    private static void ShowValue(object o)
    {
        const int HIGH_ROLL = 6;

        if (o is Dice d && d.Roll() is HIGH_ROLL)
            Console.WriteLine($"The value is {HIGH_ROLL}!");
        else
            Console.WriteLine($"The dice roll is not a {HIGH_ROLL}!");
    }
}
// The example displays output like the following:
//      The value is 6!
```

Checking for `null` can be performed using the constant pattern. The `null` keyword is supported by the `is` statement. Its syntax is:

```
expr is null
```

The following example shows a comparison of `null` checks:

```

using System;

class Program
{
    static void Main(string[] args)
    {
        object o = null;

        if (o is null)
        {
            Console.WriteLine("o does not have a value");
        }
        else
        {
            Console.WriteLine($"o is {o}");
        }

        int? x = 10;

        if (x is null)
        {
            Console.WriteLine("x does not have a value");
        }
        else
        {
            Console.WriteLine($"x is {x.Value}");
        }

        // 'null' check comparison
        Console.WriteLine($"'is' constant pattern 'null' check result : { o is null }");
        Console.WriteLine($"object.ReferenceEquals 'null' check result : { object.ReferenceEquals(o, null) }");
        Console.WriteLine($"Equality operator (==) 'null' check result : { o == null }");
    }

    // The example displays the following output:
    // o does not have a value
    // x is 10
    // 'is' constant pattern 'null' check result : True
    // object.ReferenceEquals 'null' check result : True
    // Equality operator (==) 'null' check result : True
}

```

The expression `x is null` is computed differently for reference types and nullable value types. For nullable value types, it uses `Nullable<T>.HasValue`. For reference types, it uses `(object)x == null`.

var pattern

A pattern match with the `var` pattern always succeeds. Its syntax is:

```
expr is var varname
```

Where the value of `expr` is always assigned to a local variable named `varname`. `varname` is a variable of the same type as the compile-time type of `expr`.

If `expr` evaluates to `null`, the `is` expression produces `true` and assigns `null` to `varname`. The `var` pattern is one of the few uses of `is` that produces `true` for a `null` value.

You can use the `var` pattern to create a temporary variable within a Boolean expression, as the following example shows:

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        int[] testSet = { 100271, 234335, 342439, 999683 };

        var primes = testSet.Where(n => Factor(n).ToList() is var factors
                               && factors.Count == 2
                               && factors.Contains(1)
                               && factors.Contains(n));

        foreach (int prime in primes)
        {
            Console.WriteLine($"Found prime: {prime}");
        }
    }

    static IEnumerable<int> Factor(int number)
    {
        int max = (int)Math.Sqrt(number);
        for (int i = 1; i <= max; i++)
        {
            if (number % i == 0)
            {
                yield return i;
                if (i != number / i)
                {
                    yield return number / i;
                }
            }
        }
    }
}

// The example displays the following output:
//      Found prime: 100271
//      Found prime: 999683

```

In the preceding example, the temporary variable is used to store the result of an expensive operation. The variable can then be used multiple times.

C# language specification

For more information, see [The is operator](#) section of the [C# language specification](#) and the following C# language proposals:

- [Pattern matching](#)
- [Pattern matching with generics](#)

See also

- [C# reference](#)
- [C# keywords](#)
- [Type-testing and cast operators](#)

new constraint (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `new` constraint specifies that a type argument in a generic class declaration must have a public parameterless constructor. To use the `new` constraint, the type cannot be abstract.

Apply the `new` constraint to a type parameter when a generic class creates new instances of the type, as shown in the following example:

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

When you use the `new()` constraint with other constraints, it must be specified last:

```
public class ItemFactory2<T>
    where T : IComparable, new()
{ }
```

For more information, see [Constraints on Type Parameters](#).

You can also use the `new` keyword to [create an instance of a type](#) or as a [member declaration modifier](#).

C# language specification

For more information, see the [Type parameter constraints](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Generics](#)

where (generic type constraint) (C# Reference)

3/23/2021 • 4 minutes to read • [Edit Online](#)

The `where` clause in a generic definition specifies constraints on the types that are used as arguments for type parameters in a generic type, method, delegate, or local function. Constraints can specify interfaces, base classes, or require a generic type to be a reference, value, or unmanaged type. They declare capabilities that the type argument must have.

For example, you can declare a generic class, `AGenericClass`, such that the type parameter `T` implements the `IComparable<T>` interface:

```
public class AGenericClass<T> where T : IComparable<T> { }
```

NOTE

For more information on the `where` clause in a query expression, see [where clause](#).

The `where` clause can also include a base class constraint. The base class constraint states that a type to be used as a type argument for that generic type has the specified class as a base class, or is that base class. If the base class constraint is used, it must appear before any other constraints on that type parameter. Some types are disallowed as a base class constraint: `Object`, `Array`, and `ValueType`. Before C# 7.3, `Enum`, `Delegate`, and `MulticastDelegate` were also disallowed as base class constraints. The following example shows the types that can now be specified as a base class:

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

In a nullable context in C# 8.0 and later, the nullability of the base class type is enforced. If the base class is non-nullable (for example `Base`), the type argument must be non-nullable. If the base class is nullable (for example `Base?`), the type argument may be either a nullable or non-nullable reference type. The compiler issues a warning if the type argument is a nullable reference type when the base class is non-nullable.

The `where` clause can specify that the type is a `class` or a `struct`. The `struct` constraint removes the need to specify a base class constraint of `System.ValueType`. The `System.ValueType` type may not be used as a base class constraint. The following example shows both the `class` and `struct` constraints:

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

In a nullable context in C# 8.0 and later, the `class` constraint requires a type to be a non-nullable reference type. To allow nullable reference types, use the `class?` constraint, which allows both nullable and non-nullable reference types.

The `where` clause may include the `notnull` constraint. The `notnull` constraint limits the type parameter to

non-nullable types. That type may be a [value type](#) or a non-nullable reference type. The `notnull` constraint is available starting in C# 8.0 for code compiled in a `nullable enable` context. Unlike other constraints, if a type argument violates the `notnull` constraint, the compiler generates a warning instead of an error. Warnings are only generated in a `nullable enable` context.

IMPORTANT

Generic declarations that include the `notnull` constraint can be used in a nullable oblivious context, but compiler does not enforce the constraint.

```
#nullable enable
class NotNullContainer<T>
    where T : notnull
{
}
#nullable restore
```

The `where` clause may also include an `unmanaged` constraint. The `unmanaged` constraint limits the type parameter to types known as [unmanaged types](#). The `unmanaged` constraint makes it easier to write low-level interop code in C#. This constraint enables reusable routines across all unmanaged types. The `unmanaged` constraint can't be combined with the `class` or `struct` constraint. The `unmanaged` constraint enforces that the type must be a `struct`:

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

The `where` clause may also include a constructor constraint, `new()`. That constraint makes it possible to create an instance of a type parameter using the `new` operator. The [new\(\) Constraint](#) lets the compiler know that any type argument supplied must have an accessible parameterless constructor. For example:

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

The `new()` constraint appears last in the `where` clause. The `new()` constraint can't be combined with the `struct` or `unmanaged` constraints. All types satisfying those constraints must have an accessible parameterless constructor, making the `new()` constraint redundant.

With multiple type parameters, use one `where` clause for each type parameter, for example:

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TValue>
        where TKey : IComparable<TKey>
        where TValue : IMyInterface
    {
        public void Add(TKey key, TValue val) { }
    }
}
```

You can also attach constraints to type parameters of generic methods, as shown in the following example:

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notice that the syntax to describe type parameter constraints on delegates is the same as that of methods:

```
delegate T MyDelegate<T>() where T : new();
```

For information on generic delegates, see [Generic Delegates](#).

For details on the syntax and use of constraints, see [Constraints on Type Parameters](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [new Constraint](#)
- [Constraints on Type Parameters](#)

base (C# Reference)

1/7/2020 • 2 minutes to read • [Edit Online](#)

The `base` keyword is used to access members of the base class from within a derived class:

- Call a method on the base class that has been overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the `base` keyword from within a static method.

The base class that is accessed is the base class specified in the class declaration. For example, if you specify

```
class ClassB : ClassA
```

Example

In this example, both the base class, `Person`, and the derived class, `Employee`, have a method named `GetInfo`. By using the `base` keyword, it is possible to call the `GetInfo` method on the base class, from within the derived class.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}

class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

For additional examples, see [new](#), [virtual](#), and [override](#).

Example

This example shows how to specify the base-class constructor called when creating instances of a derived class.

```

public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {

    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {

    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [this](#)

this (C# Reference)

1/7/2020 • 2 minutes to read • [Edit Online](#)

The `this` keyword refers to the current instance of the class and is also used as a modifier of the first parameter of an extension method.

NOTE

This article discusses the use of `this` with class instances. For more information about its use in extension methods, see [Extension Methods](#).

The following are common uses of `this`:

- To qualify members hidden by similar names, for example:

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- To pass an object as a parameter to other methods, for example:

```
CalcTax(this);
```

- To declare indexers, for example:

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

Static member functions, because they exist at the class level and not as part of an object, do not have a `this` pointer. It is an error to refer to `this` in a static method.

Example

In this example, `this` is used to qualify the `Employee` class members, `name` and `alias`, which are hidden by similar names. It is also used to pass an object to the method `CalcTax`, which belongs to another class.

```

class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
Name: Mingda Pan
Alias: mp
Taxes: $240.00
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)

- C# Keywords
- base
- Methods

null (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `null` keyword is a literal that represents a null reference, one that does not refer to any object. `null` is the default value of reference-type variables. Ordinary value types cannot be null, except for [nullable value types](#).

The following example demonstrates some behaviors of the `null` keyword:

```

class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# reference](#)
- [C# keywords](#)
- [Default values of C# types](#)
- [Nothing \(Visual Basic\)](#)

bool (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `bool` type keyword is an alias for the .NET `System.Boolean` structure type that represents a Boolean value, which can be either `true` or `false`.

To perform logical operations with values of the `bool` type, use [Boolean logical operators](#). The `bool` type is the result type of [comparison](#) and [equality](#) operators. A `bool` expression can be a controlling conditional expression in the `if`, `do`, `while`, and `for` statements and in the [conditional operator](#) `?:`.

The default value of the `bool` type is `false`.

Literals

You can use the `true` and `false` literals to initialize a `bool` variable or to pass a `bool` value:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

Three-valued Boolean logic

Use the nullable `bool?` type, if you need to support the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. For the `bool?` operands, the predefined `&` and `|` operators support the three-valued logic. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For more information about nullable value types, see [Nullable value types](#).

Conversions

C# provides only two conversions that involve the `bool` type. Those are an implicit conversion to the corresponding nullable `bool?` type and an explicit conversion from the `bool?` type. However, .NET provides additional methods that you can use to convert to or from the `bool` type. For more information, see the [Converting to and from Boolean values](#) section of the [System.Boolean](#) API reference page.

C# language specification

For more information, see [The bool type](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Value types](#)
- [true and false operators](#)

default (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `default` keyword can be used in two ways:

- To specify the default label in the `switch` statement.
- As the `default operator or literal` to produce the default value of a type.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

Contextual keywords (C# Reference)

4/7/2021 • 2 minutes to read • [Edit Online](#)

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#. The following contextual keywords are introduced in this section:

KEYWORD	DESCRIPTION
add	Defines a custom event accessor that is invoked when client code subscribes to the event.
and	Creates a pattern that matches when both of nested patterns match.
async	Indicates that the modified method, lambda expression, or anonymous method is asynchronous.
await	Suspends an async method until an awaited task is completed.
dynamic	Defines a reference type that enables operations in which it occurs to bypass compile-time type checking.
get	Defines an accessor method for a property or an indexer.
global	Alias of the global namespace, which is otherwise unnamed.
init	Defines an accessor method for a property or an indexer.
nint	Defines a native-sized integer data type.
not	Creates a pattern that matches when the negated pattern doesn't match.
nuint	Defines a native-sized unsigned integer data type.
or	Creates a pattern that matches when either of nested patterns matches.
partial	Defines partial classes, structs, and interfaces throughout the same compilation unit.
remove	Defines a custom event accessor that is invoked when client code unsubscribes from the event.
set	Defines an accessor method for a property or an indexer.
value	Used to set accessors and to add or remove event handlers.
var	Enables the type of a variable declared at method scope to be determined by the compiler.

KEYWORD	DESCRIPTION
<code>when</code>	Specifies a filter condition for a <code>catch</code> block or the <code>case</code> label of a <code>switch</code> statement.
<code>where</code>	Adds constraints to a generic declaration. (See also where).
<code>yield</code>	Used in an iterator block to return a value to the enumerator object or to signal the end of iteration.

All query keywords introduced in C# 3.0 are also contextual. For more information, see [Query Keywords \(LINQ\)](#).

See also

- [C# reference](#)
- [C# keywords](#)
- [C# operators and expressions](#)

add (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `add` contextual keyword is used to define a custom event accessor that is invoked when client code subscribes to your `event`. If you supply a custom `add` accessor, you must also supply a `remove` accessor.

Example

The following example shows an event that has custom `add` and `remove` accessors. For the full example, see [How to implement interface events](#).

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

You do not typically need to provide your own custom event accessors. The accessors that are automatically generated by the compiler when you declare an event are sufficient for most scenarios.

See also

- [Events](#)

get (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `get` keyword defines an *accessor* method in a property or indexer that returns the property value or the indexer element. For more information, see [Properties](#), [Auto-Implemented Properties](#) and [Indexers](#).

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Often, the `get` accessor consists of a single statement that returns a value, as it did in the previous example. Starting with C# 7.0, you can implement the `get` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessor as expression-bodied members.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Properties](#)

init (C# Reference)

3/13/2021 • 2 minutes to read • [Edit Online](#)

In C# 9 and later, the `init` keyword defines an *accessor* method in a property or indexer. An init-only setter assigns a value to the property or the indexer element only during object construction. For more information and examples, see [Properties](#), [Auto-Implemented Properties](#), and [Indexers](#).

The following example defines both a `get` and an `init` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class InitExample
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        init { _seconds = value; }
    }
}
```

Often, the `init` accessor consists of a single statement that assigns a value, as it did in the previous example. You can implement the `init` accessor as an expression-bodied member. The following example implements both the `get` and the `init` accessors as expression-bodied members.

```
class InitExampleExpressionBodied
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        init => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `init` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class InitExampleAutoProperty
{
    public double Hours { get; init; }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Properties](#)

partial type (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Partial type definitions allow for the definition of a class, struct, interface, or record to be split into multiple files.

In *File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

In *File2.cs* the declaration:

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Remarks

Splitting a class, struct or interface type over several files can be useful when you are working with large projects, or with automatically generated code such as that provided by the [Windows Forms Designer](#). A partial type may contain a [partial method](#). For more information, see [Partial Classes and Methods](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Modifiers](#)
- [Introduction to Generics](#)

partial method (C# Reference)

3/25/2021 • 2 minutes to read • [Edit Online](#)

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type. Partial methods enable class designers to provide method hooks, similar to event handlers, that developers may decide to implement or not. If the developer does not supply an implementation, the compiler removes the signature at compile time. The following conditions apply to partial methods:

- Declarations must begin with the contextual keyword [partial](#).
- Signatures in both parts of the partial type must match.

A partial method isn't required to have an implementation in the following cases:

- It doesn't have any accessibility modifiers (including the default [private](#)).
- It returns [void](#).
- It doesn't have any [out](#) parameters.
- It doesn't have any of the following modifiers [virtual](#), [override](#), [sealed](#), [new](#), or [extern](#).

Any method that doesn't conform to all those restrictions (for example, `public virtual partial void` method), must provide an implementation.

The following example shows a partial method defined in two parts of a partial class:

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Partial methods can also be useful in combination with source generators. For example a regex could be defined using the following pattern:

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

For more information, see [Partial Classes and Methods](#).

See also

- [C# Reference](#)
- [partial type](#)

remove (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `remove` contextual keyword is used to define a custom event accessor that is invoked when client code unsubscribes from your `event`. If you supply a custom `remove` accessor, you must also supply an `add` accessor.

Example

The following example shows an event with custom `add` and `remove` accessors. For the full example, see [How to implement interface events](#).

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

You do not typically need to provide your own custom event accessors. The accessors that are automatically generated by the compiler when you declare an event are sufficient for most scenarios.

See also

- [Events](#)

set (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `set` keyword defines an *accessor* method in a property or indexer that assigns a value to the property or the indexer element. For more information and examples, see [Properties](#), [Auto-Implemented Properties](#), and [Indexers](#).

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Often, the `set` accessor consists of a single statement that assigns a value, as it did in the previous example.

Starting with C# 7.0, you can implement the `set` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessors as expression-bodied members.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Properties](#)

when (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

You can use the `when` contextual keyword to specify a filter condition in the following contexts:

- In the `catch` statement of a [try/catch](#) or [try/catch/finally](#) block.
- In the `case` label of a [switch](#) statement.
- In the `switch` [expression](#).

when in a catch statement

Starting with C# 6, `when` can be used in a `catch` statement to specify a condition that must be true for the handler for a specific exception to execute. Its syntax is:

```
catch (ExceptionType [e]) when (expr)
```

where `expr` is an expression that evaluates to a Boolean value. If it returns `true`, the exception handler executes; if `false`, it does not.

The following example uses the `when` keyword to conditionally execute handlers for an [HttpRequestException](#) depending on the text of the exception message.

```

using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localhost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}

```

when in a switch statement

Starting with C# 7.0, `case` labels no longer need to be mutually exclusive, and the order in which `case` labels appear in a `switch` statement can determine which switch block executes. The `when` keyword can be used to specify a filter condition that causes its associated case label to be true only if the filter condition is also true. Its syntax is:

```
case (expr) when (when-condition):
```

where *expr* is a constant pattern or type pattern that is compared to the match expression, and *when-condition* is any Boolean expression.

The following example uses the `when` keyword to test for `Shape` objects that have an area of zero, as well as to test for a variety of `Shape` objects that have an area greater than zero.

```

using System;

public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Area = length * width;
        Circumference = 2 * length + 2 * width;
    }
}
```

```

    {
        Length = length;
        Width = width;
    }

    public double Length { get; set; }
    public double Width { get; set; }

    public override double Area
    {
        get { return Math.Round(Length * Width,2); }
    }

    public override double Circumference
    {
        get { return (Length + Width) * 2; }
    }
}

public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
        Side = side;
    }

    public double Side { get; set; }
}

public class Example
{
    public static void Main()
    {
        Shape sh = null;
        Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                           new Rectangle(10, 10), sh, new Square(0) };
        foreach (var shape in shapes)
            ShowShapeInfo(shape);
    }

    private static void ShowShapeInfo(Object obj)
    {
        switch (obj)
        {
            case Shape shape when shape.Area == 0:
                Console.WriteLine($"The shape: {shape.GetType().Name} with no dimensions");
                break;
            case Square sq when sq.Area > 0:
                Console.WriteLine("Information about the square:");
                Console.WriteLine($"    Length of a side: {sq.Side}");
                Console.WriteLine($"    Area: {sq.Area}");
                break;
            case Rectangle r when r.Area > 0:
                Console.WriteLine("Information about the rectangle:");
                Console.WriteLine($"    Dimensions: {r.Length} x {r.Width}");
                Console.WriteLine($"    Area: {r.Area}");
                break;
            case Shape shape:
                Console.WriteLine($"A {shape.GetType().Name} shape");
                break;
            case null:
                Console.WriteLine($"The {nameof(obj)} variable is uninitialized.");
                break;
            default:
                Console.WriteLine($"The {nameof(obj)} variable does not represent a Shape.");
                break;
        }
    }
}

```

```
// The example displays the following output:  
//     Information about the square:  
//         Length of a side: 10  
//         Area: 100  
//     Information about the rectangle:  
//         Dimensions: 5 x 7  
//         Area: 35  
//     Information about the rectangle:  
//         Dimensions: 10 x 10  
//         Area: 100  
//     The obj variable is uninitialized.  
//     The shape: Square with no dimensions
```

See also

- [switch statement](#)
- [try/catch statement](#)
- [try/catch/finally statement](#)

value (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The contextual keyword `value` is used in the `set` accessor in [property](#) and [indexer](#) declarations. It is similar to an input parameter of a method. The word `value` references the value that client code is attempting to assign to the property or indexer. In the following example, `MyDerivedClass` has a property called `Name` that uses the `value` parameter to assign a new string to the backing field `name`. From the point of view of client code, the operation is written as a simple assignment.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

For more information, see the [Properties](#) and [Indexers](#) articles.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

yield (C# Reference)

3/6/2021 • 4 minutes to read • [Edit Online](#)

When you use the `yield` contextual keyword in a statement, you indicate that the method, operator, or `get` accessor in which it appears is an iterator. Using `yield` to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration, see [IEnumerator<T>](#) for an example) when you implement the [IEnumerable](#) and [IEnumerator](#) pattern for a custom collection type.

The following example shows the two forms of the `yield` statement.

```
yield return <expression>;
yield break;
```

Remarks

You use a `yield return` statement to return each element one at a time.

The sequence returned from an iterator method can be consumed by using a `foreach` statement or LINQ query. Each iteration of the `foreach` loop calls the iterator method. When a `yield return` statement is reached in the iterator method, `expression` is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

You can use a `yield break` statement to end the iteration.

For more information about iterators, see [Iterators](#).

Iterator methods and get accessors

The declaration of an iterator must meet the following requirements:

- The return type must be [IEnumerable](#), [IEnumerable<T>](#), [IEnumerator](#), or [IEnumerator<T>](#).
- The declaration can't have any `in`, `ref`, or `out` parameters.

The `yield` type of an iterator that returns [IEnumerable](#) or [IEnumerator](#) is `object`. If the iterator returns [IEnumerable<T>](#) or [IEnumerator<T>](#), there must be an implicit conversion from the type of the expression in the `yield return` statement to the generic type parameter .

You can't include a `yield return` or `yield break` statement in:

- [Lambda expressions](#) and [anonymous methods](#).
- Methods that contain unsafe blocks. For more information, see [unsafe](#).

Exception handling

A `yield return` statement can't be located in a try-catch block. A `yield return` statement can be located in the try block of a try-finally statement.

A `yield break` statement can be located in a try block or a catch block but not a finally block.

If the `foreach` body (outside of the iterator method) throws an exception, a `finally` block in the iterator method is executed.

Technical implementation

The following code returns an `IEnumerable<string>` from an iterator method and then iterates through its elements.

```
IEnumerator<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
    ...
}
```

The call to `MyIteratorMethod` doesn't execute the body of the method. Instead the call returns an `IEnumerable<string>` into the `elements` variable.

On an iteration of the `foreach` loop, the `MoveNext` method is called for `elements`. This call executes the body of `MyIteratorMethod` until the next `yield return` statement is reached. The expression returned by the `yield return` statement determines not only the value of the `element` variable for consumption by the loop body but also the `Current` property of `elements`, which is an `IEnumerable<string>`.

On each subsequent iteration of the `foreach` loop, the execution of the iterator body continues from where it left off, again stopping when it reaches a `yield return` statement. The `foreach` loop completes when the end of the iterator method or a `yield break` statement is reached.

Example

The following example has a `yield return` statement that's inside a `for` loop. Each iteration of the `foreach` statement body in the `Main` method creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `yield return` statement, which occurs during the next iteration of the `for` loop.

The return type of the iterator method is `IEnumerable`, which is an iterator interface type. When the iterator method is called, it returns an enumerable object that contains the powers of a number.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

Example

The following example demonstrates a `get` accessor that is an iterator. In the example, each `yield return` statement returns an instance of a user-defined class.

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [foreach, in](#)
- [Iterators](#)

Query keywords (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section contains the contextual keywords used in query expressions.

In this section

CLAUSE	DESCRIPTION
<code>from</code>	Specifies a data source and a range variable (similar to an iteration variable).
<code>where</code>	Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators (<code>&&</code> or <code> </code>).
<code>select</code>	Specifies the type and shape that the elements in the returned sequence will have when the query is executed.
<code>group</code>	Groups query results according to a specified key value.
<code>into</code>	Provides an identifier that can serve as a reference to the results of a join, group or select clause.
<code>orderby</code>	Sorts query results in ascending or descending order based on the default comparer for the element type.
<code>join</code>	Joins two data sources based on an equality comparison between two specified matching criteria.
<code>let</code>	Introduces a range variable to store sub-expression results in a query expression.
<code>in</code>	Contextual keyword in a <code>join</code> clause.
<code>on</code>	Contextual keyword in a <code>join</code> clause.
<code>equals</code>	Contextual keyword in a <code>join</code> clause.
<code>by</code>	Contextual keyword in a <code>group</code> clause.
<code>ascending</code>	Contextual keyword in an <code>orderby</code> clause.
<code>descending</code>	Contextual keyword in an <code>orderby</code> clause.

See also

- [C# Keywords](#)
- [LINQ \(Language-Integrated Query\)](#)
- [LINQ in C#](#)

from clause (C# Reference)

11/2/2020 • 5 minutes to read • [Edit Online](#)

A query expression must begin with a `from` clause. Additionally, a query expression can contain sub-queries, which also begin with a `from` clause. The `from` clause specifies the following:

- The data source on which the query or sub-query will be run.
- A local *range variable* that represents each element in the source sequence.

Both the range variable and the data source are strongly typed. The data source referenced in the `from` clause must have a type of `IEnumerable`, `IEnumerable<T>`, or a derived type such as `IQueryable<T>`.

In the following example, `numbers` is the data source and `num` is the range variable. Note that both variables are strongly typed even though the `var` keyword is used.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
                      where num < 5
                      select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

The range variable

The compiler infers the type of the range variable when the data source implements `IEnumerable<T>`. For example, if the source has a type of `IEnumerable<Customer>`, then the range variable is inferred to be `Customer`. The only time that you must specify the type explicitly is when the source is a non-generic `IEnumerable` type such as `ArrayList`. For more information, see [How to query an ArrayList with LINQ](#).

In the previous example `num` is inferred to be of type `int`. Because the range variable is strongly typed, you can call methods on it or use it in other operations. For example, instead of writing `select num`, you could write `select num.ToString()` to cause the query expression to return a sequence of strings instead of integers. Or you could write `select num + 10` to cause the expression to return the sequence 14, 11, 13, 12, 10. For more information, see [select clause](#).

The range variable is like an iteration variable in a `foreach` statement except for one very important difference: a range variable never actually stores data from the source. It's just a syntactic convenience that enables the query to describe what will occur when the query is executed. For more information, see [Introduction to LINQ Queries \(C#\)](#).

Compound from clauses

In some cases, each element in the source sequence may itself be either a sequence or contain a sequence. For example, your data source may be an `IEnumerable<Student>` where each student object in the sequence contains a list of test scores. To access the inner list within each `student` element, you can use compound `from` clauses. The technique is like using nested `foreach` statements. You can add `where` or `orderby` clauses to either `from` clause to filter the results. The following example shows a sequence of `Student` objects, each of which contains an inner `List` of integers representing test scores. To access the inner list, use a compound `from` clause. You can insert clauses between the two `from` clauses if necessary.

```

class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
            new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
            new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
            new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
            new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                          from score in student.Scores
                          where score > 90
                          select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/

```

Using Multiple from Clauses to Perform Joins

A compound `from` clause is used to access inner collections in a single data source. However, a query can also contain multiple `from` clauses that generate supplemental queries from independent data sources. This technique enables you to perform certain types of join operations that are not possible by using the [join clause](#).

The following example shows how two `from` clauses can be used to form a complete cross join of two data sources.

```

class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Cross join:
   A is matched to x
   A is matched to y
   A is matched to z
   B is matched to x
   B is matched to y
   B is matched to z
   C is matched to x
   C is matched to y
   C is matched to z
   Filtered non-equijoin:
   y is matched to A
   y is matched to B
   y is matched to C
   z is matched to A
   z is matched to B
   z is matched to C
*/

```

For more information about join operations that use multiple `from` clauses, see [Perform left outer joins](#).

See also

- [Query Keywords \(LINQ\)](#)
- [Language Integrated Query \(LINQ\)](#)

where clause (C# Reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `where` clause is used in a query expression to specify which elements from the data source will be returned in the query expression. It applies a Boolean condition (*predicate*) to each source element (referenced by the range variable) and returns those for which the specified condition is true. A single query expression may contain multiple `where` clauses and a single clause may contain multiple predicate subexpressions.

Example

In the following example, the `where` clause filters out all numbers except those that are less than five. If you remove the `where` clause, all numbers from the data source would be returned. The expression `num < 5` is the predicate that is applied to each element.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Example

Within a single `where` clause, you can specify as many predicates as necessary by using the `&&` and `||` operators. In the following example, the query specifies two predicates in order to select only the even numbers that are less than five.

```

class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.WriteLine(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
}

// Output:
// 4 2 0
// 4 2 0

```

Example

A `where` clause may contain one or more methods that return Boolean values. In the following example, the `where` clause uses a method to determine whether the current value of the range variable is even or odd.

```

class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

        // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}

//Output: 4 8 6 2 0

```

Remarks

The `where` clause is a filtering mechanism. It can be positioned almost anywhere in a query expression, except it cannot be the first or last clause. A `where` clause may appear either before or after a `group` clause depending on whether you have to filter the source elements before or after they are grouped.

If a specified predicate is not valid for the elements in the data source, a compile-time error will result. This is one benefit of the strong type-checking provided by LINQ.

At compile time the `where` keyword is converted into a call to the `Where` Standard Query Operator method.

See also

- [Query Keywords \(LINQ\)](#)
- [from clause](#)
- [select clause](#)
- [Filtering Data](#)
- [LINQ in C#](#)
- [Language Integrated Query \(LINQ\)](#)

select clause (C# Reference)

11/2/2020 • 6 minutes to read • [Edit Online](#)

In a query expression, the `select` clause specifies the type of values that will be produced when the query is executed. The result is based on the evaluation of all the previous clauses and on any expressions in the `select` clause itself. A query expression must terminate with either a `select` clause or a `group` clause.

The following example shows a simple `select` clause in a query expression.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.WriteLine(i + " ");
        }
    }
}
//Output: 97 92 81
```

The type of the sequence produced by the `select` clause determines the type of the query variable `queryHighScores`. In the simplest case, the `select` clause just specifies the range variable. This causes the returned sequence to contain elements of the same type as the data source. For more information, see [Type Relationships in LINQ Query Operations](#). However, the `select` clause also provides a powerful mechanism for transforming (or *projecting*) source data into new types. For more information, see [Data Transformations with LINQ \(C#\)](#).

Example

The following example shows all the different forms that a `select` clause may take. In each query, note the relationship between the `select` clause and the type of the *query variable* (`studentQuery1`, `studentQuery2`, and so on).

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
        public ContactInfo GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo cInfo =
                from s in app.Students
                where s.ID == id
                select new ContactInfo
                {
                    First = s.First,
                    Last = s.Last,
                    ID = s.ID,
                    Scores = s.Scores
                };
            return cInfo;
        }
    }
}
```

```

        (from ci in app.contactList
         where ci.ID == id
         select ci)
        .FirstOrDefault();

    return cInfo;
}

public override string ToString()
{
    return First + " " + Last + ":" + ID;
}
}

public class ContactInfo
{
    public int ID { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public override string ToString() { return Email + "," + Phone; }
}

public class ScoreInfo
{
    public double Average { get; set; }
    public int ID { get; set; }
}

// The primary data source
List<Student> students = new List<Student>()
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81,
60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91,
39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},
};

// Separate data source for contact info.
List<ContactInfo> contactList = new List<ContactInfo>()
{
    new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
    new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
    new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
    new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
};

static void Main(string[] args)
{
    SelectSample2 app = new SelectSample2();

    // Produce a filtered sequence of unmodified Students.
    IEnumerable<Student> studentQuery1 =
        from student in app.students
        where student.ID > 111
        select student;

    Console.WriteLine("Query1: select range_variable");
    foreach (Student s in studentQuery1)
    {
        Console.WriteLine(s.ToString());
    }

    // Produce a filtered sequence of elements that contain
    // only one property of each Student.
    IEnumerable<String> studentQuery2 =
        from student in app.students
        where student.ID > 111

```

```

    select student.Last;

Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
foreach (string s in studentQuery2)
{
    Console.WriteLine(s);
}

// Produce a filtered sequence of objects created by
// a method call on each Student.
IEnumerable<ContactInfo> studentQuery3 =
    from student in app.students
    where student.ID > 111
    select student.GetContactInfo(app, student.ID);

Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
foreach (ContactInfo ci in studentQuery3)
{
    Console.WriteLine(ci.ToString());
}

// Produce a filtered sequence of ints from
// the internal array inside each Student.
IEnumerable<int> studentQuery4 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0];

Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
foreach (int i in studentQuery4)
{
    Console.WriteLine("First score = {0}", i);
}

// Produce a filtered sequence of doubles
// that are the result of an expression.
IEnumerable<double> studentQuery5 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0] * 1.1;

Console.WriteLine("\r\n studentQuery5: select expression");
foreach (double d in studentQuery5)
{
    Console.WriteLine("Adjusted first score = {0}", d);
}

// Produce a filtered sequence of doubles that are
// the result of a method call.
IEnumerable<double> studentQuery6 =
    from student in app.students
    where student.ID > 111
    select student.Scores.Average();

Console.WriteLine("\r\n studentQuery6: select expression2");
foreach (double d in studentQuery6)
{
    Console.WriteLine("Average = {0}", d);
}

// Produce a filtered sequence of anonymous types
// that contain only two properties from each Student.
var studentQuery7 =
    from student in app.students
    where student.ID > 111
    select new { student.First, student.Last };

Console.WriteLine("\r\n studentQuery7: select new anonymous type");
foreach (var item in studentQuery7)

```

```

    {
        Console.WriteLine("{0}, {1}", item.Last, item.First);
    }

    // Produce a filtered sequence of named objects that contain
    // a method return value and a property from each Student.
    // Use named types if you need to pass the query variable
    // across a method boundary.
    IEnumerable<ScoreInfo> studentQuery8 =
        from student in app.students
        where student.ID > 111
        select new ScoreInfo
        {
            Average = student.Scores.Average(),
            ID = student.ID
        };

    Console.WriteLine("\r\n studentQuery8: select new named type");
    foreach (ScoreInfo si in studentQuery8)
    {
        Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
    }

    // Produce a filtered sequence of students who appear on a contact list
    // and whose average is greater than 85.
    IEnumerable<ContactInfo> studentQuery9 =
        from student in app.students
        where student.Scores.Average() > 85
        join ci in app.contactList on student.ID equals ci.ID
        select ci;

    Console.WriteLine("\r\n studentQuery9: select result of join clause");
    foreach (ContactInfo ci in studentQuery9)
    {
        Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

/*
 * Output
 * Query1: select range_variable
 * Claire O'Donnell:112
 * Sven Mortensen:113
 * Cesar Garcia:114

 * studentQuery2: select range_variable.Property
 * O'Donnell
 * Mortensen
 * Garcia

 * studentQuery3: select range_variable.Method
 * ClaireO@Contoso.com,206-555-0298
 * SvenMort@Contoso.com,206-555-1130
 * CesarGar@Contoso.com,206-555-0521

 * studentQuery4: select range_variable[index]
 * First score = 75
 * First score = 88
 * First score = 97

 * studentQuery5: select expression
 * Adjusted first score = 82.5
 * Adjusted first score = 96.8
 * Adjusted first score = 106.7

 * studentQuery6: select expression?

```

```
studentQuery6: select average
Average = 72.25
Average = 84.5
Average = 88.25

studentQuery7: select new anonymous type
O'Donnell, Claire
Mortensen, Sven
Garcia, Cesar

studentQuery8: select new named type
ID = 112, Average = 72.25
ID = 113, Average = 84.5
ID = 114, Average = 88.25

studentQuery9: select result of join clause
ID = 114, Email = CesarGar@Contoso.com
*/
```

As shown in `studentQuery8` in the previous example, sometimes you might want the elements of the returned sequence to contain only a subset of the properties of the source elements. By keeping the returned sequence as small as possible you can reduce the memory requirements and increase the speed of the execution of the query. You can accomplish this by creating an anonymous type in the `select` clause and using an object initializer to initialize it with the appropriate properties from the source element. For an example of how to do this, see [Object and Collection Initializers](#).

Remarks

At compile time, the `select` clause is translated to a method call to the [Select](#) standard query operator.

See also

- [C# Reference](#)
- [Query Keywords \(LINQ\)](#)
- [from clause](#)
- [partial \(Method\) \(C# Reference\)](#)
- [Anonymous Types](#)
- [LINQ in C#](#)
- [Language Integrated Query \(LINQ\)](#)

group clause (C# Reference)

11/2/2020 • 8 minutes to read • [Edit Online](#)

The `group` clause returns a sequence of `IGrouping<TKey,TElement>` objects that contain zero or more items that match the key value for the group. For example, you can group a sequence of strings according to the first letter in each string. In this case, the first letter is the key and has a type `char`, and is stored in the `Key` property of each `IGrouping<TKey,TElement>` object. The compiler infers the type of the key.

You can end a query expression with a `group` clause, as shown in the following example:

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

If you want to perform additional query operations on each group, you can specify a temporary identifier by using the `into` contextual keyword. When you use `into`, you must continue with the query, and eventually end it with either a `select` statement or another `group` clause, as shown in the following excerpt:

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

More complete examples of the use of `group` with and without `into` are provided in the Example section of this article.

Enumerating the results of a group query

Because the `IGrouping<TKey,TElement>` objects produced by a `group` query are essentially a list of lists, you must use a nested `foreach` loop to access the items in each group. The outer loop iterates over the group keys, and the inner loop iterates over each item in the group itself. A group may have a key but no elements. The following is the `foreach` loop that executes the query in the previous code examples:

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    // Explicit type for student could also be used here.
    foreach (var student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}", student.Last, student.First);
    }
}
```

Key types

Group keys can be any type, such as a string, a built-in numeric type, or a user-defined named type or anonymous type.

Grouping by string

The previous code examples used a `char`. A string key could easily have been specified instead, for example the complete last name:

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an IEnumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

Grouping by bool

The following example shows the use of a bool value for a key to divide the results into two groups. Note that the value is produced by a sub-expression in the `group` clause.

```

class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60},},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},,
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},,
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},,
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("    {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Low averages
Omelchenko, Svetlana:77.5
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
High averages
Mortensen, Sven:93.5
Garcia, Debra:88.25
*/

```

Grouping by numeric range

The next example uses an expression to create numeric group keys that represent a percentile range. Note the

use of `let` as a convenient location to store a method call result, so that you don't have to call the method two times in the `group` clause. For more information about how to safely use methods in query expressions, see [Handle exceptions in query expressions](#).

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };
    }

    return students;
}

// This method groups students into percentile ranges based on their
// grade average. The Average method returns a double, so to produce a whole
// number it is necessary to cast to int before dividing by 10.
static void Main()
{
    // Obtain the data source.
    List<Student> students = GetStudents();

    // Write the query.
    var studentQuery =
        from student in students
        let avg = (int)student.Scores.Average()
        group student by (avg / 10) into g
        orderby g.Key
        select g;

    // Execute the query.
    foreach (var studentGroup in studentQuery)
    {
        int temp = studentGroup.Key * 10;
        Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
        foreach (var student in studentGroup)
        {
            Console.WriteLine("    {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
        }
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

/* Output:
   Students with an average between 70 and 80
   Omelchenko, Svetlana:77.5

```

```
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
Students with an average between 80 and 90
Garcia, Debra:88.25
Students with an average between 90 and 100
Mortensen, Sven:93.5
*/
```

Grouping by composite keys

Use a composite key when you want to group elements according to more than one key. You create a composite key by using an anonymous type or a named type to hold the key element. In the following example, assume that a class `Person` has been declared with members named `surname` and `city`. The `group` clause causes a separate group to be created for each set of persons with the same last name and the same city.

```
group person by new {name = person.surname, city = person.city};
```

Use a named type if you must pass the query variable to another method. Create a special class using auto-implemented properties for the keys, and then override the `Equals` and `GetHashCode` methods. You can also use a struct, in which case you do not strictly have to override those methods. For more information see [How to implement a lightweight class with auto-implemented properties](#) and [How to query for duplicate files in a directory tree](#). The latter article has a code example that demonstrates how to use a composite key with a named type.

Example

The following example shows the standard pattern for ordering source data into groups when no additional query logic is applied to the groups. This is called a grouping without a continuation. The elements in an array of strings are grouped according to their first letter. The result of the query is an `IGrouping< TKey, TElement >` type that contains a public `Key` property of type `char` and an `IEnumerable< T >` collection that contains each item in the grouping.

The result of a `group` clause is a sequence of sequences. Therefore, to access the individual elements within each returned group, use a nested `foreach` loop inside the loop that iterates the group keys, as shown in the following example.

```

class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Words that start with the letter 'b':
   blueberry
   banana
   Words that start with the letter 'c':
   chimpanzee
   cheese
   Words that start with the letter 'a':
   abacus
   apple
*/

```

Example

This example shows how to perform additional logic on the groups after you have created them, by using a *continuation* with `into`. For more information, see [into](#). The following example queries each group to select only those whose key value is a vowel.

```

class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant",
"umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("    {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Groups that start with a vowel: a
       abacus
       apple
       anteater
   Groups that start with a vowel: e
       elephant
   Groups that start with a vowel: u
       umbrella
*/

```

Remarks

At compile time, `group` clauses are translated into calls to the [GroupBy](#) method.

See also

- [IGrouping< TKey, TElement >](#)
- [GroupBy](#)
- [ThenBy](#)
- [ThenByDescending](#)
- [Query Keywords](#)
- [Language Integrated Query \(LINQ\)](#)
- [Create a nested group](#)
- [Group query results](#)
- [Perform a subquery on a grouping operation](#)

into (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `into` contextual keyword can be used to create a temporary identifier to store the results of a `group`, `join` or `select` clause into a new identifier. This identifier can itself be a generator for additional query commands. When used in a `group` or `select` clause, the use of the new identifier is sometimes referred to as a *continuation*.

Example

The following example shows the use of the `into` keyword to enable a temporary identifier `fruitGroup` which has an inferred type of `IGrouping`. By using the identifier, you can invoke the `Count` method on each group and select only those groups that contain two or more words.

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots" };

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine("{0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

The use of `into` in a `group` clause is only necessary when you want to perform additional query operations on each group. For more information, see [group clause](#).

For an example of the use of `into` in a `join` clause, see [join clause](#).

See also

- [Query Keywords \(LINQ\)](#)
- [LINQ in C#](#)
- [group clause](#)

orderby clause (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

In a query expression, the `orderby` clause causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order. Multiple keys can be specified in order to perform one or more secondary sort operations. The sorting is performed by the default comparer for the type of the element. The default sort order is ascending. You can also specify a custom comparer. However, it is only available by using method-based syntax. For more information, see [Sorting Data](#).

Example

In the following example, the first query sorts the words in alphabetical order starting from A, and second query sorts the same words in descending order. (The `ascending` keyword is the default sort value and can be omitted.)

```

class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/

```

Example

The following example performs a primary sort on the students' last names, and then a secondary sort on their first names.

```

class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }
}

```

```

public static List<Student> GetStudents()
{
    // Use a collection initializer to create the data source. Note that each element
    // in the list contains an inner sequence of scores.
    List<Student> students = new List<Student>
    {
        new Student {First="Svetlana", Last="Omelchenko", ID=111},
        new Student {First="Claire", Last="O'Donnell", ID=112},
        new Student {First="Sven", Last="Mortensen", ID=113},
        new Student {First="Cesar", Last="Garcia", ID=114},
        new Student {First="Debra", Last="Garcia", ID=115}
    };

    return students;
}
static void Main(string[] args)
{
    // Create the data source.
    List<Student> students = GetStudents();

    // Create the query.
    IEnumerable<Student> sortedStudents =
        from student in students
        orderby student.Last ascending, student.First ascending
        select student;

    // Execute the query.
    Console.WriteLine("sortedStudents:");
    foreach (Student student in sortedStudents)
        Console.WriteLine(student.Last + " " + student.First);

    // Now create groups and sort the groups. The query first sorts the names
    // of all students so that they will be in alphabetical order after they are
    // grouped. The second orderby sorts the group keys in alpha order.
    var sortedGroups =
        from student in students
        orderby student.Last, student.First
        group student by student.Last[0] into newGroup
        orderby newGroup.Key
        select newGroup;

    // Execute the query.
    Console.WriteLine(Environment.NewLine + "sortedGroups:");
    foreach (var studentGroup in sortedGroups)
    {
        Console.WriteLine(studentGroup.Key);
        foreach (var student in studentGroup)
        {
            Console.WriteLine("  {0}, {1}", student.Last, student.First);
        }
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
  Garcia, Cesar
  Garcia, Debra

```

```
Garcia, David
M
    Mortensen, Sven
O
    O'Donnell, Claire
    Omelchenko, Svetlana
*/

```

Remarks

At compile time, the `orderby` clause is translated to a call to the [OrderBy](#) method. Multiple keys in the `orderby` clause translate to [ThenBy](#) method calls.

See also

- [C# Reference](#)
- [Query Keywords \(LINQ\)](#)
- [LINQ in C#](#)
- [group clause](#)
- [Language Integrated Query \(LINQ\)](#)

join clause (C# Reference)

11/2/2020 • 9 minutes to read • [Edit Online](#)

The `join` clause is useful for associating elements from different source sequences that have no direct relationship in the object model. The only requirement is that the elements in each source share some value that can be compared for equality. For example, a food distributor might have a list of suppliers of a certain product, and a list of buyers. A `join` clause can be used, for example, to create a list of the suppliers and buyers of that product who are all in the same specified region.

A `join` clause takes two source sequences as input. The elements in each sequence must either be or contain a property that can be compared to a corresponding property in the other sequence. The `join` clause compares the specified keys for equality by using the special `equals` keyword. All joins performed by the `join` clause are equijoins. The shape of the output of a `join` clause depends on the specific type of join you are performing. The following are three most common join types:

- Inner join
- Group join
- Left outer join

Inner join

The following example shows a simple inner equijoin. This query produces a flat sequence of "product name / category" pairs. The same category string will appear in multiple elements. If an element from `categories` has no matching `products`, that category will not appear in the results.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

For more information, see [Perform inner joins](#).

Group join

A `join` clause with an `into` expression is called a group join.

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

A group join produces a hierarchical result sequence, which associates elements in the left source sequence with one or more matching elements in the right side source sequence. A group join has no equivalent in relational terms; it is essentially a sequence of object arrays.

If no elements from the right source sequence are found to match an element in the left source, the `join` clause will produce an empty array for that item. Therefore, the group join is still basically an inner-equijoin except that the result sequence is organized into groups.

If you just select the results of a group join, you can access the items, but you cannot identify the key that they

match on. Therefore, it is generally more useful to select the results of the group join into a new type that also has the key name, as shown in the previous example.

You can also, of course, use the result of a group join as the generator of another subquery:

```
var innerGroupJoinQuery2 =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from prod2 in prodGroup  
    where prod2.UnitPrice > 2.50M  
    select prod2;
```

For more information, see [Perform grouped joins](#).

Left outer join

In a left outer join, all the elements in the left source sequence are returned, even if no matching elements are in the right sequence. To perform a left outer join in LINQ, use the `DefaultIfEmpty` method in combination with a group join to specify a default right-side element to produce if a left-side element has no matches. You can use `null` as the default value for any reference type, or you can specify a user-defined default type. In the following example, a user-defined default type is shown:

```
var leftOuterJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })  
    select new { CatName = category.Name, ProdName = item.Name };
```

For more information, see [Perform left outer joins](#).

The equals operator

A `join` clause performs an equijoin. In other words, you can only base matches on the equality of two keys. Other types of comparisons such as "greater than" or "not equals" are not supported. To make clear that all joins are equijoins, the `join` clause uses the `equals` keyword instead of the `==` operator. The `equals` keyword can only be used in a `join` clause and it differs from the `==` operator in one important way. With `equals`, the left key consumes the outer source sequence, and the right key consumes the inner source. The outer source is only in scope on the left side of `equals` and the inner source sequence is only in scope on the right side.

Non-equijoins

You can perform non-equijoins, cross joins, and other custom join operations by using multiple `from` clauses to introduce new sequences independently into a query. For more information, see [Perform custom join operations](#).

Joins on object collections vs. relational tables

In a LINQ query expression, join operations are performed on object collections. Object collections cannot be "joined" in exactly the same way as two relational tables. In LINQ, explicit `join` clauses are only required when two source sequences are not tied by any relationship. When working with LINQ to SQL, foreign key tables are represented in the object model as properties of the primary table. For example, in the Northwind database, the Customer table has a foreign key relationship with the Orders table. When you map the tables to the object model, the Customer class has an Orders property that contains the collection of Orders associated with that Customer. In effect, the join has already been done for you.

For more information about querying across related tables in the context of LINQ to SQL, see [How to: Map Database Relationships](#).

Composite keys

You can test for equality of multiple values by using a composite key. For more information, see [Join by using composite keys](#). Composite keys can be also used in a `group` clause.

Example

The following example compares the results of an inner join, a group join, and a left outer join on the same data sources by using the same matching keys. Some extra code is added to these examples to clarify the results in the console display.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product {Name="Cola", CategoryID=001},
        new Product {Name="Tea", CategoryID=001},
        new Product {Name="Mustard", CategoryID=002},
        new Product {Name="Pickles", CategoryID=002},
        new Product {Name="Carrots", CategoryID=003},
        new Product {Name="Bok Choy", CategoryID=003},
        new Product {Name="Peaches", CategoryID=005},
        new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();
```

```

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

void InnerJoin()
{
    // Create the query that selects
    // a property from each element.
    var innerJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID
        select new { Category = category.ID, Product = prod.Name };

    Console.WriteLine("InnerJoin:");
    // Execute the query. Access results
    // with a simple foreach statement.
    foreach (var item in innerJoinQuery)
    {
        Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
    }
    Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin()
{
    // This is a demonstration query to show the output
    // of a "raw" group join. A more typical group join
    // is shown in the GroupInnerJoin method.
    var groupJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup;

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Simple GroupJoin:");

    // A nested foreach statement is required to access group items.
    foreach (var prodGrouping in groupJoinQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupInnerJoin()
{
    var groupJoinQuery2 =
        from category in categories
        orderby category.ID
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                       orderby prod2.Name
                       select prod2
        };
}

```

```

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupInnerJoin:");
foreach (var productGroup in groupJoinQuery2)
{
    Console.WriteLine(productGroup.Category);
    foreach (var prodItem in productGroup.Products)
    {
        totalItems++;
        Console.WriteLine(" {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
    }
}
Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin3()
{
    var groupJoinQuery3 =
        from category in categories
        join product in products on category.ID equals product.CategoryID into prodGroup
        from prod in prodGroup
        orderby prod.CategoryID
        select new { Category = prod.CategoryID, ProductName = prod.Name };

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupJoin3:");
foreach (var item in groupJoinQuery3)
{
    totalItems++;
    Console.WriteLine(" {0}:{1}", item.ProductName, item.Category);
}

Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
Console.WriteLine(System.Environment.NewLine);
}

void LeftOuterJoin()
{
    // Create the query.
    var leftOuterQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Left Outer Join:");

    // A nested foreach statement is required to access group items
    foreach (var prodGrouping in leftOuterQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

```

```

void LeftOuterJoin2()
{
    // Create the query.
    var leftOuterQuery2 =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        from item in prodGroup.DefaultIfEmpty()
        select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

    Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
    // Store the count of total items
    int totalItems = 0;

    Console.WriteLine("Left Outer Join 2:");

    // Groups have been flattened.
    foreach (var item in leftOuterQuery2)
    {
        totalItems++;
        Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
    }
    Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
}

/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:
Beverages
    Cola      1
    Tea       1
Condiments
    Mustard   2
    Pickles   2
Vegetables
    Bok Choy  3
    Carrots   3
Grains
Fruit
    Melons    5

```

```
Peaches      5
GroupInnerJoin: 8 items in 5 named groups
```

```
GroupJoin3:
  Cola:1
  Tea:1
  Mustard:2
  Pickles:2
  Carrots:3
  Bok Choy:3
  Peaches:5
  Melons:5
GroupJoin3: 8 items in 1 group
```

```
Left Outer Join:
```

```
Group:
  Cola      1
  Tea       1
Group:
  Mustard   2
  Pickles   2
Group:
  Carrots   3
  Bok Choy  3
Group:
  Nothing!  4
Group:
  Peaches   5
  Melons    5
LeftOuterJoin: 9 items in 5 groups
```

```
LeftOuterJoin2: 9 items in 1 group
```

```
Left Outer Join 2:
```

```
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
```

```
Press any key to exit.
```

```
*/
```

Remarks

A `join` clause that is not followed by `into` is translated into a [Join](#) method call. A `join` clause that is followed by `into` is translated to a [GroupJoin](#) method call.

See also

- [Query Keywords \(LINQ\)](#)
- [Language Integrated Query \(LINQ\)](#)
- [Join Operations](#)
- [group clause](#)
- [Perform left outer joins](#)
- [Perform inner joins](#)

- Perform grouped joins
- Order the results of a join clause
- Join by using composite keys
- Compatible database systems for Visual Studio

let clause (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

In a query expression, it is sometimes useful to store the result of a sub-expression in order to use it in subsequent clauses. You can do this with the `let` keyword, which creates a new range variable and initializes it with the result of the expression you supply. Once initialized with a value, the range variable cannot be used to store another value. However, if the range variable holds a queryable type, it can be queried.

Example

In the following example `let` is used in two ways:

1. To create an enumerable type that can itself be queried.
2. To enable the query to call `ToLower` only one time on the range variable `word`. Without using `let`, you would have to call `ToLower` in each predicate in the `where` clause.

```

class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("{0} starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   "A" starts with a vowel
   "is" starts with a vowel
   "a" starts with a vowel
   "earned." starts with a vowel
   "early" starts with a vowel
   "is" starts with a vowel
*/

```

See also

- [C# Reference](#)
- [Query Keywords \(LINQ\)](#)
- [LINQ in C#](#)
- [Language Integrated Query \(LINQ\)](#)
- [Handle exceptions in query expressions](#)

ascending (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `ascending` contextual keyword is used in the [orderby clause](#) in query expressions to specify that the sort order is from smallest to largest. Because `ascending` is the default sort order, you do not have to specify it.

Example

The following example shows the use of `ascending` in an [orderby clause](#).

```
IEnumerable<string> sortAscendingQuery =  
    from vegetable in vegetables  
    orderby vegetable ascending  
    select vegetable;
```

See also

- [C# Reference](#)
- [LINQ in C#](#)
- [descending](#)

descending (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `descending` contextual keyword is used in the [orderby clause](#) in query expressions to specify that the sort order is from largest to smallest.

Example

The following example shows the use of `descending` in an [orderby clause](#).

```
IEnumerable<string> sortDescendingQuery =  
    from vegetable in vegetables  
    orderby vegetable descending  
    select vegetable;
```

See also

- [C# Reference](#)
- [LINQ in C#](#)
- [ascending](#)

on (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `on` contextual keyword is used in the [join clause](#) of a query expression to specify the join condition.

Example

The following example shows the use of `on` in a `join` clause.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- [C# Reference](#)
- [Language Integrated Query \(LINQ\)](#)

equals (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `equals` contextual keyword is used in a `join` clause in a query expression to compare the elements of two sequences. For more information, see [join clause](#).

Example

The following example shows the use of the `equals` keyword in a `join` clause.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- [Language Integrated Query \(LINQ\)](#)

by (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `by` contextual keyword is used in the `group` clause in a query expression to specify how the returned items should be grouped. For more information, see [group clause](#).

Example

The following example shows the use of the `by` contextual keyword in a `group` clause to specify that the students should be grouped according to the first letter of the last name of each student.

```
var query = from student in students
            group student by student.LastName[0];
```

See also

- [LINQ in C#](#)

in (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `in` keyword is used in the following contexts:

- [generic type parameters](#) in generic interfaces and delegates.
- As a [parameter modifier](#), which lets you pass an argument to a method by reference rather than by value.
- [foreach](#) statements.
- [from clauses](#) in LINQ query expressions.
- [join clauses](#) in LINQ query expressions.

See also

- [C# Keywords](#)
- [C# Reference](#)

C# operators and expressions (C# reference)

3/6/2021 • 5 minutes to read • [Edit Online](#)

C# provides a number of operators. Many of them are supported by the [built-in types](#) and allow you to perform basic operations with values of those types. Those operators include the following groups:

- [Arithmetic operators](#) that perform arithmetic operations with numeric operands
- [Comparison operators](#) that compare numeric operands
- [Boolean logical operators](#) that perform logical operations with `bool` operands
- [Bitwise and shift operators](#) that perform bitwise or shift operations with operands of the integral types
- [Equality operators](#) that check if their operands are equal or not

Typically, you can [overload](#) those operators, that is, specify the operator behavior for the operands of a user-defined type.

The simplest C# expressions are literals (for example, [integer](#) and [real](#) numbers) and names of variables. You can combine them into complex expressions by using operators. Operator [precedence](#) and [associativity](#) determine the order in which the operations in an expression are performed. You can use parentheses to change the order of evaluation imposed by operator precedence and associativity.

In the following code, examples of expressions are at the right-hand side of assignments:

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

Typically, an expression produces a result and can be included in another expression. A `void` method call is an example of an expression that doesn't produce a result. It can be used only as a [statement](#), as the following example shows:

```
Console.WriteLine("Hello, world!");
```

Here are some other kinds of expressions that C# provides:

- [Interpolated string expressions](#) that provide convenient syntax to create formatted strings:

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- [Lambda expressions](#) that allow you to create anonymous functions:

```

int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25

```

- [Query expressions](#) that allow you to use query capabilities directly in C#:

```

var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85

```

You can use an [expression body definition](#) to provide a concise definition for a method, constructor, property, indexer, or finalizer.

Operator precedence

In an expression with multiple operators, the operators with higher precedence are evaluated before the operators with lower precedence. In the following example, the multiplication is performed first because it has higher precedence than addition:

```

var a = 2 + 2 * 2;
Console.WriteLine(a); // output: 6

```

Use parentheses to change the order of evaluation imposed by operator precedence:

```

var a = (2 + 2) * 2;
Console.WriteLine(a); // output: 8

```

The following table lists the C# operators starting with the highest precedence to the lowest. The operators within each row have the same precedence.

OPERATORS	CATEGORY OR NAME
x.y, f(x), a[i], <code>x?.y</code> , <code>x?[y]</code> , <code>x++</code> , <code>x--</code> , <code>x!</code> , <code>new</code> , <code>typeof</code> , <code>checked</code> , <code>unchecked</code> , <code>default</code> , <code>nameof</code> , <code>delegate</code> , <code>sizeof</code> , <code>stackalloc</code> , <code>x->y</code>	Primary
<code>+x</code> , <code>-x</code> , <code>!x</code> , <code>~x</code> , <code>++x</code> , <code>--x</code> , <code>^x</code> , <code>(T)x</code> , <code>await</code> , <code>&x</code> , <code>*x</code> , <code>true</code> and <code>false</code>	Unary
<code>x.y</code>	Range
<code>switch</code>	<code>switch</code> expression
<code>with</code>	<code>with</code> expression
<code>x * y</code> , <code>x / y</code> , <code>x % y</code>	Multiplicative

OPERATORS	CATEGORY OR NAME
<code>x + y, x - y</code>	Additive
<code>x << y, x >> y</code>	Shift
<code>x < y, x > y, x <= y, x >= y, is, as</code>	Relational and type-testing
<code>x == y, x != y</code>	Equality
<code>x & y</code>	Boolean logical AND or bitwise logical AND
<code>x ^ y</code>	Boolean logical XOR or bitwise logical XOR
<code>x y</code>	Boolean logical OR or bitwise logical OR
<code>x && y</code>	Conditional AND
<code>x y</code>	Conditional OR
<code>x ?? y</code>	Null-coalescing operator
<code>c ? t : f</code>	Conditional operator
<code>x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x = y, x ^= y, x <= y, x >= y, x ??= y, =></code>	Assignment and lambda declaration

Operator associativity

When operators have the same precedence, associativity of the operators determines the order in which the operations are performed:

- *Left-associative* operators are evaluated in order from left to right. Except for the [assignment operators](#) and the [null-coalescing operators](#), all binary operators are left-associative. For example, `a + b - c` is evaluated as `(a + b) - c`.
- *Right-associative* operators are evaluated in order from right to left. The assignment operators, the null-coalescing operators, and the [conditional operator](#) `?:` are right-associative. For example, `x = y = z` is evaluated as `x = (y = z)`.

Use parentheses to change the order of evaluation imposed by operator associativity:

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}"); // output: a = 1, b = 6
```

Operand evaluation

Unrelated to operator precedence and associativity, operands in an expression are evaluated from left to right. The following examples demonstrate the order in which operators and operands are evaluated:

EXPRESSION	ORDER OF EVALUATION
a + b	a, b, +
a + b * c	a, b, c, *, +
a / b + c * d	a, b, /, c, d, *, +
a / (b + c) * d	a, b, c, +, /, d, *

Typically, all operator operands are evaluated. However, some operators evaluate operands conditionally. That is, the value of the leftmost operand of such an operator defines if (or which) other operands should be evaluated. These operators are the conditional logical **AND** (`&&`) and **OR** (`||`) operators, the **null-coalescing operators** `??` and `??:=`, the **null-conditional operators** `?.` and `?[]`, and the **conditional operator** `?::`. For more information, see the description of each operator.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Expressions](#)
- [Operators](#)

See also

- [C# reference](#)
- [Operator overloading](#)
- [Expression trees](#)

Arithmetic operators (C# reference)

11/2/2020 • 9 minutes to read • [Edit Online](#)

The following operators perform arithmetic operations with operands of numeric types:

- Unary `++` (increment), `--` (decrement), `+` (plus), and `-` (minus) operators
- Binary `*` (multiplication), `/` (division), `%` (remainder), `+` (addition), and `-` (subtraction) operators

Those operators are supported by all [integral](#) and [floating-point](#) numeric types.

In the case of integral types, those operators (except the `++` and `--` operators) are defined for the `int`, `uint`, `long`, and `ulong` types. When operands are of other integral types (`sbyte`, `byte`, `short`, `ushort`, or `char`), their values are converted to the `int` type, which is also the result type of an operation. When operands are of different integral or floating-point types, their values are converted to the closest containing type, if such a type exists. For more information, see the [Numeric promotions](#) section of the [C# language specification](#). The `++` and `--` operators are defined for all integral and floating-point numeric types and the `char` type.

Increment operator `++`

The unary increment operator `++` increments its operand by 1. The operand must be a variable, a [property](#) access, or an [indexer](#) access.

The increment operator is supported in two forms: the postfix increment operator, `x++`, and the prefix increment operator, `++x`.

Postfix increment operator

The result of `x++` is the value of `x` *before* the operation, as the following example shows:

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i++);
Console.WriteLine(i);    // output: 4
```

Prefix increment operator

The result of `++x` is the value of `x` *after* the operation, as the following example shows:

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);    // output: 2.5
```

Decrement operator `--`

The unary decrement operator `--` decrements its operand by 1. The operand must be a variable, a [property](#) access, or an [indexer](#) access.

The decrement operator is supported in two forms: the postfix decrement operator, `x--`, and the prefix decrement operator, `--x`.

Postfix decrement operator

The result of `x--` is the value of `x` *before* the operation, as the following example shows:

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);    // output: 2
```

Prefix decrement operator

The result of `--x` is the value of `x` *after* the operation, as the following example shows:

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);    // output: 0.5
```

Unary plus and minus operators

The unary `+` operator returns the value of its operand. The unary `-` operator computes the numeric negation of its operand.

```
Console.WriteLine(+4);      // output: 4

Console.WriteLine(-4);      // output: -4
Console.WriteLine(-(-4)); // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);        // output: -5
Console.WriteLine(b.GetType()); // output: System.Int64

Console.WriteLine(-double.NaN); // output: NaN
```

The `ulong` type doesn't support the unary `-` operator.

Multiplication operator `*`

The multiplication operator `*` computes the product of its operands:

```
Console.WriteLine(5 * 2);      // output: 10
Console.WriteLine(0.5 * 2.5);    // output: 1.25
Console.WriteLine(0.1m * 23.4m); // output: 2.34
```

The unary `*` operator is the [pointer indirection operator](#).

Division operator `/`

The division operator `/` divides its left-hand operand by its right-hand operand.

Integer division

For the operands of integer types, the result of the `/` operator is of an integer type and equals the quotient of the two operands rounded towards zero:

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

To obtain the quotient of the two operands as a floating-point number, use the `float`, `double`, or `decimal` type:

```
Console.WriteLine(13 / 5.0);      // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b); // output: 2.6
```

Floating-point division

For the `float`, `double`, and `decimal` types, the result of the `/` operator is the quotient of the two operands:

```
Console.WriteLine(16.8f / 4.1f);  // output: 4.097561
Console.WriteLine(16.8d / 4.1d);  // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);  // output: 4.09756097560975609756098
```

If one of the operands is `decimal`, another operand can be neither `float` nor `double`, because neither `float` nor `double` is implicitly convertible to `decimal`. You must explicitly convert the `float` or `double` operand to the `decimal` type. For more information about conversions between numeric types, see [Built-in numeric conversions](#).

Remainder operator %

The remainder operator `%` computes the remainder after dividing its left-hand operand by its right-hand operand.

Integer remainder

For the operands of integer types, the result of `a % b` is the value produced by `a - (a / b) * b`. The sign of the non-zero remainder is the same as that of the left-hand operand, as the following example shows:

```
Console.WriteLine(5 % 4);    // output: 1
Console.WriteLine(5 % -4);   // output: 1
Console.WriteLine(-5 % 4);   // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Use the `Math.DivRem` method to compute both integer division and remainder results.

Floating-point remainder

For the `float` and `double` operands, the result of `x % y` for the finite `x` and `y` is the value `z` such that

- The sign of `z`, if non-zero, is the same as the sign of `x`.
- The absolute value of `z` is the value produced by `|x| - n * |y|` where `n` is the largest possible integer that is less than or equal to `|x| / |y|` and `|x|` and `|y|` are the absolute values of `x` and `y`, respectively.

NOTE

This method of computing the remainder is analogous to that used for integer operands, but different from the IEEE 754 specification. If you need the remainder operation that complies with the IEEE 754 specification, use the `Math.IEEEremainder` method.

For information about the behavior of the `%` operator with non-finite operands, see the [Remainder operator](#) section of the [C# language specification](#).

For the `decimal` operands, the remainder operator `%` is equivalent to the [remainder operator](#) of the `System.Decimal` type.

The following example demonstrates the behavior of the remainder operator with floating-point operands:

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m); // output: 2.8
```

Addition operator +

The addition operator `+` computes the sum of its operands:

```
Console.WriteLine(5 + 4);      // output: 9
Console.WriteLine(5 + 4.3);    // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

You can also use the `+` operator for string concatenation and delegate combination. For more information, see the [+ and += operators](#) article.

Subtraction operator -

The subtraction operator `-` subtracts its right-hand operand from its left-hand operand:

```
Console.WriteLine(47 - 3);     // output: 44
Console.WriteLine(5 - 4.3);    // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

You can also use the `-` operator for delegate removal. For more information, see the [- and -= operators](#) article.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with arithmetic operators:

```
int a = 5;
a += 9;
Console.WriteLine(a); // output: 14

a -= 4;
Console.WriteLine(a); // output: 10

a *= 2;
Console.WriteLine(a); // output: 20

a /= 4;
Console.WriteLine(a); // output: 5

a %= 3;
Console.WriteLine(a); // output: 2
```

Because of [numeric promotions](#), the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType()); // output: System.Int32
Console.WriteLine(c); // output: 300

a += b;
Console.WriteLine(a); // output: 44
```

You also use the `+=` and `-=` operators to subscribe to and unsubscribe from an [event](#), respectively. For more information, see [How to subscribe to and unsubscribe from events](#).

Operator precedence and associativity

The following list orders arithmetic operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators
- Prefix increment `++x` and decrement `--x` and unary `+` and `-` operators
- Multiplicative `*`, `/`, and `%` operators
- Additive `+` and `-` operators

Binary arithmetic operators are left-associative. That is, operators with the same precedence level are evaluated from left to right.

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence and associativity.

```
Console.WriteLine(2 + 2 * 2); // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2); // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Arithmetic overflow and division by zero

When the result of an arithmetic operation is outside the range of possible finite values of the involved numeric type, the behavior of an arithmetic operator depends on the type of its operands.

Integer arithmetic overflow

Integer division by zero always throws a [DivideByZeroException](#).

In case of integer arithmetic overflow, an overflow checking context, which can be [checked](#) or [unchecked](#), controls the resulting behavior:

- In a checked context, if overflow happens in a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run time, an [OverflowException](#) is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that don't fit in the destination type.

Along with the [checked](#) and [unchecked](#) statements, you can use the `checked` and `unchecked` operators to control the overflow checking context, in which an expression is evaluated:

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b)); // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

By default, arithmetic operations occur in an *unchecked* context.

Floating-point arithmetic overflow

Arithmetic operations with the `float` and `double` types never throw an exception. The result of arithmetic operations with those types can be one of special values that represent infinity and not-a-number:

```
double a = 1.0 / 0.0;
Console.WriteLine(a); // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b); // output: NaN
Console.WriteLine(double.IsNaN(b)); // output: True
```

For the operands of the `decimal` type, arithmetic overflow always throws an [OverflowException](#) and division by zero always throws a [DivideByZeroException](#).

Round-off errors

Because of general limitations of the floating-point representation of real numbers and floating-point arithmetic, round-off errors might occur in calculations with floating-point types. That is, the produced result of an expression might differ from the expected mathematical result. The following example demonstrates several such cases:

For more information, see remarks at the [System.Double](#), [System.Single](#), or [System.Decimal](#) reference pages.

Operator overloadability

A user-defined type can **overload** the unary (+ +, - -, +, and -) and binary (*, /, %, +, and -) arithmetic operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type cannot explicitly overload a compound assignment operator.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- Postfix increment and decrement operators
 - Prefix increment and decrement operators
 - Unary plus operator
 - Unary minus operator
 - Multiplication operator
 - Division operator
 - Remainder operator
 - Addition operator
 - Subtraction operator
 - Compound assignment
 - The checked and unchecked operators
 - Numeric promotions

See also

- C# reference
 - C# operators and expressions
 - System.Math
 - System.MathF
 - Numerics in .NET

Boolean logical operators (C# reference)

11/2/2020 • 7 minutes to read • [Edit Online](#)

The following operators perform logical operations with `bool` operands:

- Unary `!` (logical negation) operator.
- Binary `&` (logical AND), `|` (logical OR), and `^` (logical exclusive OR) operators. Those operators always evaluate both operands.
- Binary `&&` (conditional logical AND) and `||` (conditional logical OR) operators. Those operators evaluate the right-hand operand only if it's necessary.

For operands of the [integral numeric types](#), the `&`, `|`, and `^` operators perform bitwise logical operations. For more information, see [Bitwise and shift operators](#).

Logical negation operator !

The unary prefix `!` operator computes logical negation of its operand. That is, it produces `true`, if the operand evaluates to `false`, and `false`, if the operand evaluates to `true`:

```
bool passed = false;
Console.WriteLine(!passed); // output: True
Console.WriteLine(!true); // output: False
```

Beginning with C# 8.0, the unary postfix `!` operator is the [null-forgiving operator](#).

Logical AND operator &

The `&` operator computes the logical AND of its operands. The result of `x & y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`.

The `&` operator evaluates both operands even if the left-hand operand evaluates to `false`, so that the operation result is `false` regardless of the value of the right-hand operand.

In the following example, the right-hand operand of the `&` operator is a method call, which is performed regardless of the value of the left-hand operand:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [conditional logical AND operator](#) `&&` also computes the logical AND of its operands, but doesn't evaluate the right-hand operand if the left-hand operand evaluates to `false`.

For operands of the [integral numeric types](#), the `&` operator computes the [bitwise logical AND](#) of its operands. The unary `&` operator is the [address-of operator](#).

Logical exclusive OR operator ^

The `^` operator computes the logical exclusive OR, also known as the logical XOR, of its operands. The result of `x ^ y` is `true` if `x` evaluates to `true` and `y` evaluates to `false`, or `x` evaluates to `false` and `y` evaluates to `true`. Otherwise, the result is `false`. That is, for the `bool` operands, the `^` operator computes the same result as the [inequality operator](#) `!=`.

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

For operands of the [integral numeric types](#), the `^` operator computes the [bitwise logical exclusive OR](#) of its operands.

Logical OR operator |

The `|` operator computes the logical OR of its operands. The result of `x | y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`.

The `|` operator evaluates both operands even if the left-hand operand evaluates to `true`, so that the operation result is `true` regardless of the value of the right-hand operand.

In the following example, the right-hand operand of the `|` operator is a method call, which is performed regardless of the value of the left-hand operand:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [conditional logical OR operator](#) `||` also computes the logical OR of its operands, but doesn't evaluate the right-hand operand if the left-hand operand evaluates to `true`.

For operands of the [integral numeric types](#), the `|` operator computes the [bitwise logical OR](#) of its operands.

Conditional logical AND operator &&

The conditional logical AND operator `&&`, also known as the "short-circuiting" logical AND operator, computes the logical AND of its operands. The result of `x && y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`. If `x` evaluates to `false`, `y` is not evaluated.

In the following example, the right-hand operand of the `&&` operator is a method call, which isn't performed if the left-hand operand evaluates to `false`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [logical AND operator](#) `&` also computes the logical AND of its operands, but always evaluates both operands.

Conditional logical OR operator `||`

The conditional logical OR operator `||`, also known as the "short-circuiting" logical OR operator, computes the logical OR of its operands. The result of `x || y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`. If `x` evaluates to `true`, `y` is not evaluated.

In the following example, the right-hand operand of the `||` operator is a method call, which isn't performed if the left-hand operand evaluates to `true`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [logical OR operator](#) `|` also computes the logical OR of its operands, but always evaluates both operands.

Nullable Boolean logical operators

For `bool?` operands, the `&` (logical AND) and `|` (logical OR) operators support the three-valued logic as follows:

- The `&` operator produces `true` only if both its operands evaluate to `true`. If either `x` or `y` evaluates to `false`, `x & y` produces `false` (even if another operand evaluates to `null`). Otherwise, the result of `x & y` is `null`.
- The `|` operator produces `false` only if both its operands evaluate to `false`. If either `x` or `y` evaluates to `true`, `x | y` produces `true` (even if another operand evaluates to `null`). Otherwise, the result of `x | y` is `null`.

The following table presents that semantics:

<code>X</code>	<code>Y</code>	<code>X&Y</code>	<code>X Y</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>null</code>	<code>null</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>null</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>true</code>	<code>null</code>	<code>true</code>
<code>null</code>	<code>false</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>

The behavior of those operators differs from the typical operator behavior with nullable value types. Typically, an operator which is defined for operands of a value type can be also used with operands of the corresponding nullable value type. Such an operator produces `null` if any of its operands evaluates to `null`. However, the `&` and `|` operators can produce non-null even if one of the operands evaluates to `null`. For more information about the operator behavior with nullable value types, see the [Lifted operators](#) section of the [Nullable value types](#) article.

You can also use the `!` and `^` operators with `bool?` operands, as the following example shows:

```
bool? test = null;
Display(!test);           // output: null
Display(test ^ false);   // output: null
Display(test ^ null);    // output: null
Display(true ^ null);    // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

The conditional logical operators `&&` and `||` don't support `bool?` operands.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

The `&`, `|`, and `^` operators support compound assignment, as the following example shows:

```
bool test = true;
test &= false;
Console.WriteLine(test); // output: False

test |= true;
Console.WriteLine(test); // output: True

test ^= false;
Console.WriteLine(test); // output: True
```

NOTE

The conditional logical operators `&&` and `||` don't support compound assignment.

Operator precedence

The following list orders logical operators starting from the highest precedence to the lowest:

- Logical negation operator `!`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`
- Conditional logical AND operator `&&`
- Conditional logical OR operator `||`

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence:

```
Console.WriteLine(true | true & false); // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Operator overloadability

A user-defined type can [overload](#) the `!`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type cannot explicitly overload a compound assignment operator.

A user-defined type cannot overload the conditional logical operators `&&` and `||`. However, if a user-defined type overloads the [true and false operators](#) and the `&` or `|` operator in a certain way, the `&&` or `||` operation, respectively, can be evaluated for the operands of that type. For more information, see the [User-defined conditional logical operators](#) section of the [C# language specification](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Logical negation operator](#)
- [Logical operators](#)
- [Conditional logical operators](#)
- [Compound assignment](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Bitwise and shift operators](#)

Bitwise and shift operators (C# reference)

11/2/2020 • 7 minutes to read • [Edit Online](#)

The following operators perform bitwise or shift operations with operands of the [integral numeric types](#) or the [char](#) type:

- Unary `~` (bitwise complement) operator
- Binary `<<` (left shift) and `>>` (right shift) shift operators
- Binary `&` (logical AND), `|` (logical OR), and `^` (logical exclusive OR) operators

Those operators are defined for the `int`, `uint`, `long`, and `ulong` types. When both operands are of other integral types (`sbyte`, `byte`, `short`, `ushort`, or `char`), their values are converted to the `int` type, which is also the result type of an operation. When operands are of different integral types, their values are converted to the closest containing integral type. For more information, see the [Numeric promotions](#) section of the [C# language specification](#).

The `&`, `|`, and `^` operators are also defined for operands of the `bool` type. For more information, see [Boolean logical operators](#).

Bitwise and shift operations never cause overflow and produce the same results in [checked](#) and [unchecked](#) contexts.

Bitwise complement operator `~`

The `~` operator produces a bitwise complement of its operand by reversing each bit:

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

You can also use the `~` symbol to declare finalizers. For more information, see [Finalizers](#).

Left-shift operator `<<`

The `<<` operator shifts its left-hand operand left by the [number of bits defined by its right-hand operand](#).

The left-shift operation discards the high-order bits that are outside the range of the result type and sets the low-order empty bit positions to zero, as the following example shows:

```
uint x = 0b_1100_1001_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 110010010000000000000000000000010001
// After: 1001000000000000000000000000000100010001
```

Because the shift operators are defined only for the `int`, `uint`, `long`, and `ulong` types, the result of an operation always contains at least 32 bits. If the left-hand operand is of another integral type (`sbyte`, `byte`,

`short`, `ushort`, or `char`), its value is converted to the `int` type, as the following example shows:

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

For information about how the right-hand operand of the `<<` operator defines the shift count, see the [Shift count of the shift operators](#) section.

Right-shift operator `>>`

The `>>` operator shifts its left-hand operand right by the [number of bits defined by its right-hand operand](#).

The right-shift operation discards the low-order bits, as the following example shows:

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

The high-order empty bit positions are set based on the type of the left-hand operand as follows:

- If the left-hand operand is of type `int` or `long`, the right-shift operator performs an *arithmetic* shift: the value of the most significant bit (the sign bit) of the left-hand operand is propagated to the high-order empty bit positions. That is, the high-order empty bit positions are set to zero if the left-hand operand is non-negative and set to one if it's negative.

```
int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After: {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After:   11110000000000000000000000000000
```

- If the left-hand operand is of type `uint` or `ulong`, the right-shift operator performs a *logical* shift: the high-order empty bit positions are always set to zero.

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After: {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:    10000000000000000000000000000000
```

For information about how the right-hand operand of the `>>` operator defines the shift count, see the [Shift](#)

count of the shift operators section.

Logical AND operator &

The `&` operator computes the bitwise logical AND of its integral operands:

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

For `bool` operands, the `&` operator computes the [logical AND](#) of its operands. The unary `&` operator is the [address-of operator](#).

Logical exclusive OR operator ^

The `^` operator computes the bitwise logical exclusive OR, also known as the bitwise logical XOR, of its integral operands:

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

For `bool` operands, the `^` operator computes the [logical exclusive OR](#) of its operands.

Logical OR operator |

The `|` operator computes the bitwise logical OR of its integral operands:

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

For `bool` operands, the `|` operator computes the [logical OR](#) of its operands.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with bitwise and shift operators:

```
uint a = 0b_1111_1000;
a &= 0b_1001_1101;
Display(a); // output: 10011000

a |= 0b_0011_0001;
Display(a); // output: 10111001

a ^= 0b_1000_0000;
Display(a); // output: 111001

a <<= 2;
Display(a); // output: 11100100

a >>= 4;
Display(a); // output: 1110

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");
```

Because of [numeric promotions](#), the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}"); // output: 1111000100000000

x <<= 8;
Console.WriteLine(x); // output: 0
```

Operator precedence

The following list orders bitwise and shift operators starting from the highest precedence to the lowest:

- Bitwise complement operator `~`
- Shift operators `<<` and `>>`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence:

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1); // output: 1101

uint d2 = (a | b) & c;
Display(d2); // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");
```

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the

Shift count of the shift operators

For the shift operators `<<` and `>>`, the type of the right-hand operand must be `int` or a type that has a [predefined implicit numeric conversion](#) to `int`.

For the `x << count` and `x >> count` expressions, the actual shift count depends on the type of `x` as follows:

- If the type of `x` is `int` or `uint`, the shift count is defined by the low-order *five* bits of the right-hand operand. That is, the shift count is computed from `count & 0x1F` (or `count & 0b_1_1111`).
- If the type of `x` is `long` or `ulong`, the shift count is defined by the low-order *six* bits of the right-hand operand. That is, the shift count is computed from `count & 0x3F` (or `count & 0b_11_1111`).

The following example demonstrates that behavior:

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

NOTE

As the preceding example shows, the result of a shift operation can be non-zero even if the value of the right-hand operand is greater than the number of bits in the left-hand operand.

Enumeration logical operators

The `~`, `&`, `|`, and `^` operators are also supported by any [enumeration](#) type. For operands of the same enumeration type, a logical operation is performed on the corresponding values of the underlying integral type. For example, for any `x` and `y` of an enumeration type `T` with an underlying type `U`, the `x & y` expression produces the same result as the `(T)((U)x & (U)y)` expression.

You typically use bitwise logical operators with an enumeration type that is defined with the [Flags](#) attribute. For more information, see the [Enumeration types as bit flags](#) section of the [Enumeration types](#) article.

Operator overloadability

A user-defined type can [overload](#) the `~`, `<<`, `>>`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type cannot explicitly overload a compound assignment operator.

If a user-defined type `T` overloads the `<<` or `>>` operator, the type of the left-hand operand must be `T` and the type of the right-hand operand must be `int`.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Bitwise complement operator](#)
- [Shift operators](#)
- [Logical operators](#)
- [Compound assignment](#)
- [Numeric promotions](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Boolean logical operators](#)

Equality operators (C# reference)

11/2/2020 • 4 minutes to read • [Edit Online](#)

The `==` (equality) and `!=` (inequality) operators check if their operands are equal or not.

Equality operator `==`

The equality operator `==` returns `true` if its operands are equal, `false` otherwise.

Value types equality

Operands of the [built-in value types](#) are equal if their values are equal:

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b); // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2); // output: False
Console.WriteLine(c1 == char.ToLower(c2)); // output: True
```

NOTE

For the `==`, `<`, `>`, `<=`, and `>=` operators, if any of the operands is not a number ([Double.NaN](#) or [Single.NaN](#)), the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value, including `NaN`. For more information and examples, see the [Double.NaN](#) or [Single.NaN](#) reference article.

Two operands of the same [enum](#) type are equal if the corresponding values of the underlying integral type are equal.

User-defined [struct](#) types don't support the `==` operator by default. To support the `==` operator, a user-defined struct must [overload](#) it.

Beginning with C# 7.3, the `==` and `!=` operators are supported by C# [tuples](#). For more information, see the [Tuple equality](#) section of the [Tuple types](#) article.

Reference types equality

By default, two non-record reference-type operands are equal if they refer to the same object:

```

public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b); // output: False
        Console.WriteLine(a == c); // output: True
    }
}

```

As the example shows, user-defined reference types support the `==` operator by default. However, a reference type can overload the `==` operator. If a reference type overloads the `==` operator, use the [Object.ReferenceEquals](#) method to check if two references of that type refer to the same object.

Record types equality

Available in C# 9.0 and later, [record types](#) support the `==` and `!=` operators that by default provide value equality semantics. That is, two record operands are equal when both of them are `null` or corresponding values of all fields and auto-implemented properties are equal.

```

public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2); // output: False
        Console.WriteLine(p1 == p3); // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2); // output: False
    }
}

```

As the preceding example shows, in case of non-record reference-type members their reference values are compared, not the referenced instances.

String equality

Two [string](#) operands are equal when both of them are `null` or both string instances are of the same length and have identical characters in each character position:

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower()); // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3); // output: False
```

That is a case-sensitive ordinal comparison. For more information about string comparison, see [How to compare strings in C#](#).

Delegate equality

Two [delegate](#) operands of the same runtime type are equal when both of them are `null` or their invocation lists are of the same length and have equal entries in each position:

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c)); // output: False
Console.WriteLine(b == c); // output: True
```

For more information, see the [Delegate equality operators](#) section of the [C# language specification](#).

Delegates that are produced from evaluation of semantically identical [lambda expressions](#) are not equal, as the following example shows:

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b); // output: False
Console.WriteLine(a + b == a + b); // output: True
Console.WriteLine(b + a == a + b); // output: False
```

Inequality operator !=

The inequality operator `!=` returns `true` if its operands are not equal, `false` otherwise. For the operands of the [built-in types](#), the expression `x != y` produces the same result as the expression `!(x == y)`. For more information about type equality, see the [Equality operator](#) section.

The following example demonstrates the usage of the `!=` operator:

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b); // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2); // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2); // output: True
```

Operator overloadability

A user-defined type can [overload](#) the `==` and `!=` operators. If a type overloads one of the two operators, it

must also overload the other one.

A record type cannot explicitly overload the `==` and `!=` operators. If you need to change the behavior of the `==` and `!=` operators for record type `T`, implement the `IEquatable<T>.Equals` method with the following signature:

```
public virtual bool Equals(T? other);
```

C# language specification

For more information, see the [Relational and type-testing operators](#) section of the [C# language specification](#).

For more information about equality of record types, see the [Equality members](#) section of the [records feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [System.IEquatable<T>](#)
- [Object.Equals](#)
- [Object.ReferenceEquals](#)
- [Equality comparisons](#)
- [Comparison operators](#)

Comparison operators (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal) comparison, also known as relational, operators compare their operands. Those operators are supported by all [integral](#) and [floating-point](#) numeric types.

NOTE

For the `==`, `<`, `>`, `<=`, and `>=` operators, if any of the operands is not a number ([Double.NaN](#) or [Single.NaN](#)), the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value, including `NaN`. For more information and examples, see the [Double.NaN](#) or [Single.NaN](#) reference article.

The `char` type also supports comparison operators. In the case of `char` operands, the corresponding character codes are compared.

Enumeration types also support comparison operators. For operands of the same `enum` type, the corresponding values of the underlying integral type are compared.

The `==` and `!=` operators check if their operands are equal or not.

Less than operator <

The `<` operator returns `true` if its left-hand operand is less than its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 < 5.1);    // output: False
Console.WriteLine(5.1 < 5.1);    // output: False
Console.WriteLine(0.0 < 5.1);    // output: True

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

Greater than operator >

The `>` operator returns `true` if its left-hand operand is greater than its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 > 5.1);    // output: True
Console.WriteLine(5.1 > 5.1);    // output: False
Console.WriteLine(0.0 > 5.1);    // output: False

Console.WriteLine(double.NaN > 5.1);    // output: False
Console.WriteLine(double.NaN <= 5.1);    // output: False
```

Less than or equal operator <=

The `<=` operator returns `true` if its left-hand operand is less than or equal to its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 <= 5.1);    // output: False
Console.WriteLine(5.1 <= 5.1);    // output: True
Console.WriteLine(0.0 <= 5.1);    // output: True

Console.WriteLine(double.NaN > 5.1);    // output: False
Console.WriteLine(double.NaN <= 5.1);    // output: False
```

Greater than or equal operator `>=`

The `>=` operator returns `true` if its left-hand operand is greater than or equal to its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 >= 5.1);    // output: True
Console.WriteLine(5.1 >= 5.1);    // output: True
Console.WriteLine(0.0 >= 5.1);    // output: False

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

Operator overloadability

A user-defined type can [overload](#) the `<`, `>`, `<=`, and `>=` operators.

If a type overloads one of the `<` or `>` operators, it must overload both `<` and `>`. If a type overloads one of the `<=` or `>=` operators, it must overload both `<=` and `>=`.

C# language specification

For more information, see the [Relational and type-testing operators](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [System.IComparable<T>](#)
- [Equality operators](#)

Member access operators and expressions (C# reference)

3/23/2021 • 7 minutes to read • [Edit Online](#)

You can use the following operators and expressions when you access a type member:

- `.` (member access): to access a member of a namespace or a type
- `[]` (array element or indexer access): to access an array element or a type indexer
- `?.` and `?[]` (null-conditional operators): to perform a member or element access operation only if an operand is non-null
- `()` (invocation): to call an accessed method or invoke a delegate
- `^` (index from end): to indicate that the element position is from the end of a sequence
- `..` (range): to specify a range of indices that you can use to obtain a range of sequence elements

Member access expression .

You use the `.` token to access a member of a namespace or a type, as the following examples demonstrate:

- Use `.` to access a nested namespace within a namespace, as the following example of a `using` directive shows:

```
using System.Collections.Generic;
```

- Use `.` to form a *qualified name* to access a type within a namespace, as the following code shows:

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

Use a `using` directive to make the use of qualified names optional.

- Use `.` to access `type members`, static and non-static, as the following code shows:

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

You can also use `.` to access an `extension method`.

Indexer operator []

Square brackets, `[]`, are typically used for array, indexer, or pointer element access.

Array access

The following example demonstrates how to access array elements:

```

int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]); // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant); // output: -3

```

If an array index is outside the bounds of the corresponding dimension of an array, an [IndexOutOfRangeException](#) is thrown.

As the preceding example shows, you also use square brackets when you declare an array type or instantiate an array instance.

For more information about arrays, see [Arrays](#).

Indexer access

The following example uses the .NET [Dictionary<TKey,TValue>](#) type to demonstrate indexer access:

```

var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]); // output: 4.14159265358979

```

Indexers allow you to index instances of a user-defined type in the similar way as array indexing. Unlike array indices, which must be integer, the indexer parameters can be declared to be of any type.

For more information about indexers, see [Indexers](#).

Other usages of []

For information about pointer element access, see the [Pointer element access operator \[\]](#) section of the [Pointer related operators](#) article.

You also use square brackets to specify [attributes](#):

```

[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}

```

Null-conditional operators ?. and ?[]

Available in C# 6 and later, a null-conditional operator applies a [member access](#), `?.`, or [element access](#), `?[]`, operation to its operand only if that operand evaluates to non-null; otherwise, it returns `null`. That is,

- If `a` evaluates to `null`, the result of `a?.x` or `a?[]` is `null`.
- If `a` evaluates to non-null, the result of `a?.x` or `a?[]` is the same as the result of `a.x` or `a[]`, respectively.

NOTE

If `a.x` or `a[x]` throws an exception, `a?.x` or `a?[x]` would throw the same exception for non-null `a`. For example, if `a` is a non-null array instance and `x` is outside the bounds of `a`, `a?[x]` would throw an `IndexOutOfRangeException`.

The null-conditional operators are short-circuiting. That is, if one operation in a chain of conditional member or element access operations returns `null`, the rest of the chain doesn't execute. In the following example, `B` is not evaluated if `A` evaluates to `null` and `C` is not evaluated if `A` or `B` evaluates to `null`:

```
A?.B?.Do(C);  
A?.B?[C];
```

The following example demonstrates the usage of the `?.` and `?[]` operators:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)  
{  
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;  
}  
  
var sum1 = SumNumbers(null, 0);  
Console.WriteLine(sum1); // output: NaN  
  
var numberSets = new List<double[]>  
{  
    new[] { 1.0, 2.0, 3.0 },  
    null  
};  
  
var sum2 = SumNumbers(numberSets, 0);  
Console.WriteLine(sum2); // output: 6  
  
var sum3 = SumNumbers(numberSets, 1);  
Console.WriteLine(sum3); // output: NaN
```

The preceding example also uses the **null-coalescing operator** `??` to specify an alternative expression to evaluate in case the result of a null-conditional operation is `null`.

If `a.x` or `a[x]` is of a non-nullable value type `T`, `a?.x` or `a?[x]` is of the corresponding **nullable value type** `T?`. If you need an expression of type `T`, apply the null-coalescing operator `??` to a null-conditional expression, as the following example shows:

```
int GetSumOfFirstTwoOrDefault(int[] numbers)  
{  
    if ((numbers?.Length ?? 0) < 2)  
    {  
        return 0;  
    }  
    return numbers[0] + numbers[1];  
}  
  
Console.WriteLine(GetSumOfFirstTwoOrDefault(null)); // output: 0  
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0])); // output: 0  
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 })); // output: 7
```

In the preceding example, if you don't use the `??` operator, `numbers?.Length < 2` evaluates to `false` when `numbers` is `null`.

The null-conditional member access operator `?.` is also known as the Elvis operator.

Thread-safe delegate invocation

Use the `?.` operator to check if a delegate is non-null and invoke it in a thread-safe way (for example, when you [raise an event](#)), as the following code shows:

```
PropertyChanged?.Invoke(...)
```

That code is equivalent to the following code that you would use in C# 5 or earlier:

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(...);
}
```

That is a thread-safe way to ensure that only a non-null `handler` is invoked. Because delegate instances are immutable, no thread can change the object referenced by the `handler` local variable. In particular, if the code executed by another thread unsubscribes from the `PropertyChanged` event and `PropertyChanged` becomes `null` before `handler` is invoked, the object referenced by `handler` remains unaffected. The `?.` operator evaluates its left-hand operand no more than once, guaranteeing that it cannot be changed to `null` after being verified as non-null.

Invocation expression ()

Use parentheses, `()`, to call a [method](#) or invoke a [delegate](#).

The following example demonstrates how to call a method, with or without arguments, and invoke a delegate:

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count); // output: 2

numbers.Clear();
display(numbers.Count); // output: 0
```

You also use parentheses when you invoke a [constructor](#) with the `new` operator.

Other usages of ()

You also use parentheses to adjust the order in which to evaluate operations in an expression. For more information, see [C# operators](#).

[Cast expressions](#), which perform explicit type conversions, also use parentheses.

Index from end operator ^

Available in C# 8.0 and later, the `^` operator indicates the element position from the end of a sequence. For a sequence of length `length`, `^n` points to the element with offset `length - n` from the start of a sequence. For example, `^1` points to the last element of a sequence and `^length` points to the first element of a sequence.

```

int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last); // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast); // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first); // output: T

```

As the preceding example shows, expression `^e` is of the [System.Index](#) type. In expression `^e`, the result of `e` must be implicitly convertible to `int`.

You can also use the `^` operator with the [range operator](#) to create a range of indices. For more information, see [Indices and ranges](#).

Range operator ..

Available in C# 8.0 and later, the `..` operator specifies the start and end of a range of indices as its operands.

The left-hand operand is an *inclusive* start of a range. The right-hand operand is an *exclusive* end of a range. Either of operands can be an index from the start or from the end of a sequence, as the following example shows:

```

int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset); // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner); // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end); // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));

```

As the preceding example shows, expression `a..b` is of the [System.Range](#) type. In expression `a..b`, the results of `a` and `b` must be implicitly convertible to `int` or [Index](#).

You can omit any of the operands of the `..` operator to obtain an open-ended range:

- `a..` is equivalent to `a..^0`
- `..b` is equivalent to `0..b`
- `..` is equivalent to `0..^0`

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf); // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf); // output: 0 10 20

int[] all = numbers[...];
Display(all); // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

For more information, see [Indices and ranges](#).

Operator overloadability

The `.`, `()`, `^`, and `..` operators cannot be overloaded. The `[]` operator is also considered a non-overloadable operator. Use [indexers](#) to support indexing with user-defined types.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Member access](#)
- [Element access](#)
- [Null-conditional operator](#)
- [Invocation expressions](#)

For more information about indices and ranges, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [?? \(null-coalescing operator\)](#)
- [:: operator](#)

Type-testing operators and cast expression (C# reference)

11/2/2020 • 6 minutes to read • [Edit Online](#)

You can use the following operators and expressions to perform type checking or type conversion:

- [is operator](#): to check if the runtime type of an expression is compatible with a given type
- [as operator](#): to explicitly convert an expression to a given type if its runtime type is compatible with that type
- [cast expression](#): to perform an explicit conversion
- [typeof operator](#): to obtain the `System.Type` instance for a type

is operator

The `E is T` operator checks if the runtime type of an expression result is compatible with a given type. Beginning with C# 7.0, the `is` operator also tests an expression result against a pattern.

The expression with the type-testing `is` operator has the following form

```
E is T
```

where `E` is an expression that returns a value and `T` is the name of a type or a type parameter. `E` cannot be an anonymous method or a lambda expression.

The `E is T` expression returns `true` if the result of `E` is non-null and can be converted to type `T` by a reference conversion, a boxing conversion, or an unboxing conversion; otherwise, it returns `false`. The `is` operator doesn't consider user-defined conversions.

The following example demonstrates that the `is` operator returns `true` if the runtime type of an expression result derives from a given type, that is, there exists a reference conversion between types:

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base); // output: True
        Console.WriteLine(b is Derived); // output: False

        object d = new Derived();
        Console.WriteLine(d is Base); // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

The next example shows that the `is` operator takes into account boxing and unboxing conversions but doesn't consider [numeric conversions](#):

```
int i = 27;
Console.WriteLine(i is System.IFormattable); // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int); // output: True
Console.WriteLine(iBoxed is long); // output: False
```

For information about C# conversions, see the [Conversions](#) chapter of the [C# language specification](#).

Type testing with pattern matching

Beginning with C# 7.0, the `is` operator also tests an expression result against a pattern. In particular, it supports the type pattern in the following form:

```
E is T v
```

where `E` is an expression that returns a value, `T` is the name of a type or a type parameter, and `v` is a new local variable of type `T`. If the result of `E` is non-null and can be converted to `T` by a reference, boxing, or unboxing conversion, the `E is T v` expression returns `true` and the converted value of the result of `E` is assigned to variable `v`.

The following example demonstrates the usage of the `is` operator with the type pattern:

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 30
}
```

For more information about the type pattern and other supported patterns, see [Pattern matching with is](#).

as operator

The `as` operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion is not possible, the `as` operator returns `null`. Unlike a [cast expression](#), the `as` operator never throws an exception.

The expression of the form

```
E as T
```

where `E` is an expression that returns a value and `T` is the name of a type or a type parameter, produces the same result as

```
E is T ? (T)(E) : (T)null
```

except that `E` is only evaluated once.

The `as` operator considers only reference, nullable, boxing, and unboxing conversions. You cannot use the `as` operator to perform a user-defined conversion. To do that, use a [cast expression](#).

The following example demonstrates the usage of the `as` operator:

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]); // output: 40
}
```

NOTE

As the preceding example shows, you need to compare the result of the `as` expression with `null` to check if the conversion is successful. Beginning with C# 7.0, you can use the `is operator` both to test if the conversion succeeds and, if it succeeds, assign its result to a new variable.

Cast expression

A cast expression of the form `(T)E` performs an explicit conversion of the result of expression `E` to type `T`. If no explicit conversion exists from the type of `E` to type `T`, a compile-time error occurs. At run time, an explicit conversion might not succeed and a cast expression might throw an exception.

The following example demonstrates explicit numeric and reference conversions:

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a); // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count); // output: 3
Console.WriteLine(list[1]); // output: 20
```

For information about supported explicit conversions, see the [Explicit conversions](#) section of the [C# language specification](#). For information about how to define a custom explicit or implicit type conversion, see [User-defined conversion operators](#).

Other usages of ()

You also use parentheses to [call a method or invoke a delegate](#).

Other use of parentheses is to adjust the order in which to evaluate operations in an expression. For more information, see [C# operators](#).

typeof operator

The `typeof` operator obtains the `System.Type` instance for a type. The argument to the `typeof` operator must be the name of a type or a type parameter, as the following example shows:

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

You can also use the `typeof` operator with unbound generic types. The name of an unbound generic type must contain the appropriate number of commas, which is one less than the number of type parameters. The following example shows the usage of the `typeof` operator with an unbound generic type:

```
Console.WriteLine(typeof(Dictionary<, >));  
// Output:  
// System.Collections.Generic.Dictionary`2[TKey, TValue]
```

An expression cannot be an argument of the `typeof` operator. To get the [System.Type](#) instance for the runtime type of an expression result, use the [Object.GetType](#) method.

Type testing with the `typeof` operator

Use the `typeof` operator to check if the runtime type of the expression result exactly matches a given type. The following example demonstrates the difference between type checking performed with the `typeof` operator and the `is` operator:

```
public class Animal { }  
  
public class Giraffe : Animal { }  
  
public static class TypeOfExample  
{  
    public static void Main()  
    {  
        object b = new Giraffe();  
        Console.WriteLine(b is Animal); // output: True  
        Console.WriteLine(b.GetType() == typeof(Animal)); // output: False  
  
        Console.WriteLine(b is Giraffe); // output: True  
        Console.WriteLine(b.GetType() == typeof(Giraffe)); // output: True  
    }  
}
```

Operator overloadability

The `is`, `as`, and `typeof` operators cannot be overloaded.

A user-defined type cannot overload the `()` operator, but can define custom type conversions that can be performed by a cast expression. For more information, see [User-defined conversion operators](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [The `is` operator](#)
- [The `as` operator](#)
- [Cast expressions](#)
- [The `typeof` operator](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [How to safely cast by using pattern matching and the `is` and `as` operators](#)
- [Generics in .NET](#)

User-defined conversion operators (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A user-defined type can define a custom implicit or explicit conversion from or to another type.

Implicit conversions don't require special syntax to be invoked and can occur in a variety of situations, for example, in assignments and methods invocations. Predefined C# implicit conversions always succeed and never throw an exception. User-defined implicit conversions should behave in that way as well. If a custom conversion can throw an exception or lose information, define it as an explicit conversion.

User-defined conversions are not considered by the `is` and `as` operators. Use a [cast expression](#) to invoke a user-defined explicit conversion.

Use the `operator` and `implicit` or `explicit` keywords to define an implicit or explicit conversion, respectively. The type that defines a conversion must be either a source type or a target type of that conversion. A conversion between two user-defined types can be defined in either of the two types.

The following example demonstrates how to define an implicit and explicit conversion:

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number); // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit); // output: 7
    }
}
```

You also use the `operator` keyword to overload a predefined C# operator. For more information, see [Operator overloading](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Conversion operators](#)
- [User-defined conversions](#)
- [Implicit conversions](#)
- [Explicit conversions](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Operator overloading](#)
- [Type-testing and cast operators](#)
- [Casting and type conversion](#)
- [Design guidelines - Conversion operators](#)
- [Chained user-defined explicit conversions in C#](#)

Pointer related operators (C# reference)

4/6/2021 • 7 minutes to read • [Edit Online](#)

You can use the following operators to work with pointers:

- Unary `&` (address-of) operator: to get the address of a variable
- Unary `*` (pointer indirection) operator: to obtain the variable pointed by a pointer
- The `->` (member access) and `[]` (element access) operators
- Arithmetic operators `+`, `-`, `++`, and `--`
- Comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`

For information about pointer types, see [Pointer types](#).

NOTE

Any operation with pointers requires an `unsafe` context. The code that contains unsafe blocks must be compiled with the [AllowUnsafeBlocks](#) compiler option.

Address-of operator &

The unary `&` operator returns the address of its operand:

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

The operand of the `&` operator must be a fixed variable. *Fixed* variables are variables that reside in storage locations that are unaffected by operation of the [garbage collector](#). In the preceding example, the local variable `number` is a fixed variable, because it resides on the stack. Variables that reside in storage locations that can be affected by the garbage collector (for example, relocated) are called *movable* variables. Object fields and array elements are examples of movable variables. You can get the address of a movable variable if you "fix", or "pin", it with a [fixed statement](#). The obtained address is valid only inside the block of a `fixed` statement. The following example shows how to use a `fixed` statement and the `&` operator:

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

You can't get the address of a constant or a value.

For more information about fixed and movable variables, see the [Fixed and moveable variables](#) section of the [C# language specification](#).

The binary `&` operator computes the [logical AND](#) of its Boolean operands or the [bitwise logical AND](#) of its integral operands.

Pointer indirection operator `*`

The unary pointer indirection operator `*` obtains the variable to which its operand points. It's also known as the dereference operator. The operand of the `*` operator must be of a pointer type.

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

You cannot apply the `*` operator to an expression of type `void*`.

The binary `*` operator computes the [product](#) of its numeric operands.

Pointer member access operator `->`

The `->` operator combines [pointer indirection](#) and [member access](#). That is, if `x` is a pointer of type `T*` and `y` is an accessible member of type `T`, an expression of the form

```
x->y
```

is equivalent to

```
(*x).y
```

The following example demonstrates the usage of the `->` operator:

```

public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString()); // output: (3, 4)
    }
}

```

You cannot apply the `->` operator to an expression of type `void*`.

Pointer element access operator []

For an expression `p` of a pointer type, a pointer element access of the form `p[n]` is evaluated as `*(p + n)`, where `n` must be of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`. For information about the behavior of the `+` operator with pointers, see the [Addition or subtraction of an integral value to or from a pointer](#) section.

The following example demonstrates how to access array elements with a pointer and the `[]` operator:

```

unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

In the preceding example, a `stackalloc` expression allocates a block of memory on the stack.

NOTE

The pointer element access operator doesn't check for out-of-bounds errors.

You cannot use `[]` for pointer element access with an expression of type `void*`.

You can also use the `[]` operator for [array element or indexer access](#).

Pointer arithmetic operators

You can perform the following arithmetic operations with pointers:

- Add or subtract an integral value to or from a pointer
- Subtract two pointers
- Increment or decrement a pointer

You cannot perform those operations with pointers of type `void*`.

For information about supported arithmetic operations with numeric types, see [Arithmetic operators](#).

Addition or subtraction of an integral value to or from a pointer

For a pointer `p` of type `T*` and an expression `n` of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`, addition and subtraction are defined as follows:

- Both `p + n` and `n + p` expressions produce a pointer of type `T*` that results from adding `n * sizeof(T)` to the address given by `p`.
- The `p - n` expression produces a pointer of type `T*` that results from subtracting `n * sizeof(T)` from the address given by `p`.

The `sizeof` operator obtains the size of a type in bytes.

The following example demonstrates the usage of the `+` operator with a pointer:

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Pointer subtraction

For two pointers `p1` and `p2` of type `T*`, the expression `p1 - p2` produces the difference between the addresses given by `p1` and `p2` divided by `sizeof(T)`. The type of the result is `long`. That is, `p1 - p2` is computed as `((long)(p1) - (long)(p2)) / sizeof(T)`.

The following example demonstrates the pointer subtraction:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1); // output: 4
}
```

Pointer increment and decrement

The `++` increment operator adds 1 to its pointer operand. The `--` decrement operator subtracts 1 from its pointer operand.

Both operators are supported in two forms: postfix (`p++` and `p--`) and prefix (`++p` and `--p`). The result of `p++` and `p--` is the value of `p` *before* the operation. The result of `++p` and `--p` is the value of `p` *after* the operation.

The following example demonstrates the behavior of both postfix and prefix increment operators:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Pointer comparison operators

You can use the `==`, `!=`, `<`, `>`, `<=`, and `>=` operators to compare operands of any pointer type, including `void*`. Those operators compare the addresses given by the two operands as if they were unsigned integers.

For information about the behavior of those operators for operands of other types, see the [Equality operators](#) and [Comparison operators](#) articles.

Operator precedence

The following list orders pointer related operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators and the `->` and `[]` operators
- Prefix increment `++x` and decrement `--x` operators and the `&` and `*` operators
- Additive `+` and `-` operators
- Comparison `<`, `>`, `<=`, and `>=` operators
- Equality `==` and `!=` operators

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence.

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Operator overloadability

A user-defined type cannot overload the pointer related operators `&`, `*`, `->`, and `[]`.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Fixed and moveable variables](#)
- [The address-of operator](#)
- [Pointer indirection](#)

- [Pointer member access](#)
- [Pointer element access](#)
- [Pointer arithmetic](#)
- [Pointer increment and decrement](#)
- [Pointer comparison](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Pointer types](#)
- [unsafe keyword](#)
- [fixed keyword](#)
- [stackalloc](#)
- [sizeof operator](#)

Assignment operators (C# reference)

4/7/2021 • 2 minutes to read • [Edit Online](#)

The assignment operator `=` assigns the value of its right-hand operand to a variable, a [property](#), or an [indexer](#) element given by its left-hand operand. The result of an assignment expression is the value assigned to the left-hand operand. The type of the right-hand operand must be the same as the type of the left-hand operand or implicitly convertible to it.

The assignment operator `=` is right-associative, that is, an expression of the form

```
a = b = c
```

is evaluated as

```
a = (b = c)
```

The following example demonstrates the usage of the assignment operator with a local variable, a property, and an indexer element as its left-hand operand:

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

ref assignment operator

Beginning with C# 7.3, you can use the `ref` assignment operator `= ref` to reassign a [ref local](#) or [ref readonly local](#) variable. The following example demonstrates the usage of the `ref` assignment operator:

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

In the case of the `ref` assignment operator, both of its operands must be of the same type.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

Compound assignment is supported by [arithmetic](#), [Boolean logical](#), and [bitwise logical and shift](#) operators.

Null-coalescing assignment

Beginning with C# 8.0, you can use the null-coalescing assignment operator `??=` to assign the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`. For more information, see the [?? and ??= operators](#) article.

Operator overloadability

A user-defined type cannot [overload](#) the assignment operator. However, a user-defined type can define an implicit conversion to another type. That way, the value of a user-defined type can be assigned to a variable, a property, or an indexer element of another type. For more information, see [User-defined conversion operators](#).

A user-defined type cannot explicitly overload a compound assignment operator. However, if a user-defined type overloads a binary operator `op`, the `op=` operator, if it exists, is also implicitly overloaded.

C# language specification

For more information, see the [Assignment operators](#) section of the [C# language specification](#).

For more information about the `ref` assignment operator `= ref`, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [ref keyword](#)

Lambda expressions (C# reference)

3/6/2021 • 11 minutes to read • [Edit Online](#)

You use a *lambda expression* to create an anonymous function. Use the [lambda declaration operator](#) `=>` to separate the lambda's parameter list from its body. A lambda expression can be of any of the following two forms:

- [Expression lambda](#) that has an expression as its body:

```
(input-parameters) => expression
```

- [Statement lambda](#) that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Any lambda expression can be converted to a [delegate](#) type. The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value. If a lambda expression doesn't return a value, it can be converted to one of the [Action](#) delegate types; otherwise, it can be converted to one of the [Func](#) delegate types. For example, a lambda expression that has two parameters and returns no value can be converted to an [Action<T1,T2>](#) delegate. A lambda expression that has one parameter and returns a value can be converted to a [Func<T,TResult>](#) delegate. In the following example, the lambda expression `x => x * x`, which specifies a parameter that's named `x` and returns the value of `x` squared, is assigned to a variable of a delegate type:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Expression lambdas can also be converted to the [expression tree](#) types, as the following example shows:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

You can use lambda expressions in any code that requires instances of delegate types or expression trees, for example as an argument to the [Task.Run\(Action\)](#) method to pass the code that should be executed in the background. You can also use lambda expressions when you write [LINQ in C#](#), as the following example shows:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

When you use method-based syntax to call the `Enumerable.Select` method in the `System.Linq.Enumerable` class, for example in LINQ to Objects and LINQ to XML, the parameter is a delegate type `System.Func<T,TResult>`.

When you call the `Queryable.Select` method in the `System.Linq.Queryable` class, for example in LINQ to SQL, the parameter type is an expression tree type `Expression<Func<TSource, TResult>>`. In both cases you can use the same lambda expression to specify the parameter value. That makes the two `Select` calls to look similar although in fact the type of objects created from the lambdas is different.

Expression lambdas

A lambda expression with an expression on the right side of the `=>` operator is called an *expression lambda*. An expression lambda returns the result of the expression and takes the following basic form:

```
(input-parameters) => expression
```

The body of an expression lambda can consist of a method call. However, if you are creating `expression trees` that are evaluated outside the context of the .NET Common Language Runtime (CLR), such as in SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET Common Language Runtime (CLR).

Statement lambdas

A statement lambda resembles an expression lambda except that its statements are enclosed in braces:

```
(input-parameters) => { <sequence-of-statements> }
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

You cannot use statement lambdas to create expression trees.

Input parameters of a lambda expression

You enclose input parameters of a lambda expression in parentheses. Specify zero input parameters with empty parentheses:

```
Action line = () => Console.WriteLine();
```

If a lambda expression has only one input parameter, parentheses are optional:

```
Func<double, double> cube = x => x * x * x;
```

Two or more input parameters are separated by commas:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Sometimes the compiler can't infer the types of input parameters. You can specify the types explicitly as shown in the following example:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Input parameter types must be all explicit or all implicit; otherwise, a [CS0748](#) compiler error occurs.

Beginning with C# 9.0, you can use [discards](#) to specify two or more input parameters of a lambda expression that aren't used in the expression:

```
Func<int, int, int> constant = (_, _) => 42;
```

Lambda discard parameters may be useful when you use a lambda expression to [provide an event handler](#).

NOTE

For backwards compatibility, if only a single input parameter is named `_`, then, within a lambda expression, `_` is treated as the name of that parameter.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the [async](#) and [await](#) keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

You can add the same event handler by using an `async` lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows:

```

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}

```

For more information about how to create and use `async` methods, see [Asynchronous Programming with `async` and `await`](#).

Lambda expressions and tuples

Starting with C# 7.0, the C# language provides built-in support for [tuples](#). You can provide a tuple as an argument to a lambda expression, and your lambda expression can also return a tuple. In some cases, the C# compiler uses type inference to determine the types of tuple components.

You define a tuple by enclosing a comma-delimited list of its components in parentheses. The following example uses tuple with three components to pass a sequence of numbers to a lambda expression, which doubles each value and returns a tuple with three components that contains the result of the multiplications.

```

Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)

```

Ordinarily, the fields of a tuple are named `Item1`, `Item2`, etc. You can, however, define a tuple with named components, as the following example does.

```

Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");

```

For more information about C# tuples, see [Tuple types](#).

Lambdas with the standard query operators

LINQ to Objects, among other implementations, have an input parameter whose type is one of the `Func<TResult>` family of generic delegates. These delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. `Func` delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the `Func<T,TResult>` delegate type:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

The delegate can be instantiated as a `Func<int, bool>` instance where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. For example,

`Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when it's invoked, returns Boolean value that indicates whether the input parameter is equal to five:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result); // False
```

You can also supply a lambda expression when the argument type is an `Expression<TDelegate>`, for example in the standard query operators that are defined in the `Queryable` type. When you specify an `Expression<TDelegate>` argument, the lambda is compiled to an expression tree.

The following example uses the `Count` standard query operator:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following example produces a sequence that contains all elements in the `numbers` array that precede the 9, because that's the first number in the sequence that doesn't meet the condition:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

The following example specifies multiple input parameters by enclosing them in parentheses. The method returns all the elements in the `numbers` array until it encounters a number whose value is less than its ordinal position in the array:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Type inference in lambda expressions

When writing lambdas, you often don't have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter types, and other factors as described in the C# language specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. If you are querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for type inference for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Note that lambda expressions in themselves don't have a type because the common type system has no intrinsic concept of "lambda expression." However, it's sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or [Expression](#) type to which the lambda expression is converted.

Capture of outer variables and variable scope in lambda expressions

Lambdas can refer to *outer variables*. These are the variables that are in scope in the method that defines the lambda expression, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```

public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"'{j}' is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
    }
}

```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression are not visible in the enclosing method.
- A lambda expression cannot directly capture an `in`, `ref`, or `out` parameter from the enclosing method.
- A `return` statement in a lambda expression doesn't cause the enclosing method to return.
- A lambda expression cannot contain a `goto`, `break`, or `continue` statement if the target of that jump statement is outside the lambda expression block. It's also an error to have a jump statement outside the

lambda expression block if the target is inside the block.

Beginning with C# 9.0, you can apply the `static` modifier to a lambda expression to prevent unintentional capture of local variables or instance state by the lambda:

```
Func<double, double> square = static x => x * x;
```

A static lambda can't capture local variables or instance state from enclosing scopes, but may reference static members and constant definitions.

C# language specification

For more information, see the [Anonymous function expressions](#) section of the [C# language specification](#).

For more information about features added in C# 9.0, see the following feature proposal notes:

- [Lambda discard parameters](#)
- [Static anonymous functions](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [LINQ \(Language-Integrated Query\)](#)
- [Expression Trees](#)
- [Local functions vs. lambda expressions](#)
- [Visual Studio 2008 C# Samples \(see LINQ Sample Queries files and XQuery program\)](#)

Patterns (C# reference)

4/6/2021 • 14 minutes to read • [Edit Online](#)

C# introduced pattern matching in C# 7.0. Since then, each major C# version extends pattern matching capabilities. The following C# expressions and statements support pattern matching:

- `is expression`
- `switch statement`
- `switch expression` (introduced in C# 8.0)

In those constructs, you can match an input expression against any of the following patterns:

- **Declaration pattern**: to check the runtime type of an expression and, if a match succeeds, assign an expression result to a declared variable. Introduced in C# 7.0.
- **Type pattern**: to check the runtime type of an expression. Introduced in C# 9.0.
- **Constant pattern**: to test if an expression result equals a specified constant. Introduced in C# 7.0.
- **Relational patterns**: to compare an expression result with a specified constant. Introduced in C# 9.0.
- **Logical patterns**: to test if an expression matches a logical combination of patterns. Introduced in C# 9.0.
- **Property pattern**: to test if an expression's properties or fields match nested patterns. Introduced in C# 8.0.
- **Positional pattern**: to deconstruct an expression result and test if the resulting values match nested patterns. Introduced in C# 8.0.
- `var pattern`: to match any expression and assign its result to a declared variable. Introduced in C# 7.0.
- **Discard pattern**: to match any expression. Introduced in C# 8.0.

Logical, **property**, and **positional** patterns are *recursive* patterns. That is, they can contain *nested* patterns.

For the example of how to use those patterns to build a data-driven algorithm, see [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#).

Declaration and type patterns

You use declaration and type patterns to check if the runtime type of an expression is compatible with a given type. With a declaration pattern, you can also declare a new local variable. When a declaration pattern matches an expression, that variable is assigned a converted expression result, as the following example shows:

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower()); // output: hello, world!
}
```

Beginning with C# 7.0, a *declaration pattern* with type `T` matches an expression when an expression result is non-null and any of the following conditions are true:

- The runtime type of an expression result is `T`.
- The runtime type of an expression result derives from type `T` or implements interface `T` or another [implicit reference conversion](#) exists from it to `T`. The following example demonstrates two cases when this condition is true:

```

var numbers = new int[] { 10, 20, 30 };
Console.WriteLine(GetSourceLabel(numbers)); // output: 1

var letters = new List<char> { 'a', 'b', 'c', 'd' };
Console.WriteLine(GetSourceLabel(letters)); // output: 2

static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
{
    Array array => 1,
    ICollection<T> collection => 2,
    _ => 3,
};

```

In the preceding example, at the first call to the `GetSourceLabel` method, the first pattern matches an argument value because the argument's runtime type `int[]` derives from the `Array` type. At the second call to the `GetSourceLabel` method, the argument's runtime type `List<T>` doesn't derive from the `Array` type but implements the `ICollection<T>` interface.

- The runtime type of an expression result is a `nullable value type` with the underlying type `T`.
- A `boxing` or `unboxing` conversion exists from the runtime type of an expression result to type `T`.

The following example demonstrates the last two conditions:

```

int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b); // output: 30
}

```

If you want to check only the type of an expression, you can use a discard `_` in place of a variable's name, as the following example shows:

```

public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
    };
}

```

Beginning with C# 9.0, for that purpose you can use a *type pattern*, as the following example shows:

```

public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
};

```

Like a declaration pattern, a type pattern matches an expression when an expression result is non-null and its runtime type satisfies any of the conditions listed above.

For more information, see the [Declaration pattern](#) and [Type pattern](#) sections of the feature proposal notes.

Constant pattern

Beginning with C# 7.0, you use a *constant pattern* to test if an expression result equals a specified constant, as the following example shows:

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors: {visitorCount}",
        nameof(visitorCount)),
};
```

In a constant pattern, you can use any constant expression, such as:

- an [integer](#) or [floating-point](#) numerical literal
- a [char](#) or a [string](#) literal
- a Boolean value `true` or `false`
- an [enum](#) value
- the name of a declared [const](#) field or local
- `null`

Use a constant pattern to check for `null`, as the following example shows:

```
if (input is null)
{
    return;
}
```

The compiler guarantees that no user-overloaded equality operator `==` is invoked when expression `x is null` is evaluated.

Beginning with C# 9.0, you can use a [negated](#) `null` constant pattern to check for non-null, as the following example shows:

```
if (input is not null)
{
    // ...
}
```

For more information, see the [Constant pattern](#) section of the feature proposal note.

Relational patterns

Beginning with C# 9.0, you use a *relational pattern* to compare an expression result with a constant, as the following example shows:

```

Console.WriteLine(Classify(13)); // output: Too high
Console.WriteLine(Classify(double.NaN)); // output: Unknown
Console.WriteLine(Classify(2.4)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};

```

In a relational pattern, you can use any of the [relational operators](#) `<`, `>`, `<=`, or `>=`. The right-hand part of a relational pattern must be a constant expression. The constant expression can be of an [integer](#), [floating-point](#), [char](#), or [enum](#) type.

To check if an expression result is in a certain range, match it against a [conjunctive](#) `and` [pattern](#), as the following example shows:

```

Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14))); // output: spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19))); // output: summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17))); // output: winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};

```

If an expression result is `null` or fails to convert to the type of a constant by a nullable or unboxing conversion, a relational pattern doesn't match an expression.

For more information, see the [Relational patterns](#) section of the feature proposal note.

Logical patterns

Beginning with C# 9.0, you use the `not`, `and`, and `or` pattern combinator to create the following *logical patterns*:

- *Negation* `not` pattern that matches an expression when the negated pattern doesn't match the expression. The following example shows how you can negate a [constant](#) `null` pattern to check if an expression is non-null:

```

if (input is not null)
{
    // ...
}

```

- *Conjunctive* `and` pattern that matches an expression when both patterns match the expression. The following example shows how you can combine [relational patterns](#) to check if a value is in a certain range:

```

Console.WriteLine(Classify(13)); // output: High
Console.WriteLine(Classify(-100)); // output: Too low
Console.WriteLine(Classify(5.7)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -40.0 => "Too low",
    >= -40.0 and < 0 => "Low",
    >= 0 and < 10.0 => "Acceptable",
    >= 10.0 and < 20.0 => "High",
    >= 20.0 => "Too high",
    double.NaN => "Unknown",
};

```

- *Disjunctive* `or` pattern that matches an expression when either of patterns matches the expression, as the following example shows:

```

Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19))); // output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9))); // output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11))); // output: spring

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};

```

As the preceding example shows, you can repeatedly use the pattern combinators in a pattern.

The `and` pattern combinator has higher precedence than `or`. To explicitly specify the precedence, use parentheses, as the following example shows:

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

NOTE

The order in which patterns are checked is undefined. At run time, the right-hand nested patterns of `or` and `and` patterns can be checked first.

For more information, see the [Pattern combinators](#) section of the feature proposal note.

Property pattern

Beginning with C# 8.0, you use a *property pattern* to match an expression's properties or fields against nested patterns, as the following example shows:

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month: 5, Day: 19 or 20 or 21 };
```

A property pattern matches an expression when an expression result is non-null and every nested pattern matches the corresponding property or field of the expression result.

You can also add a runtime type check and a variable declaration to a property pattern, as the following example

shows:

```
Console.WriteLine(TakeFive("Hello, world!")); // output: Hello
Console.WriteLine(TakeFive("Hi!")); // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' })); // output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' })); // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new string(symbols.Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

A property pattern is a recursive pattern. That is, you can use any pattern as a nested pattern. Use a property pattern to match parts of data against nested patterns, as the following example shows:

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndAtOrigin(Segment segment) =>
    segment is { Start: { X: 0, Y: 0 } } or { End: { X: 0, Y: 0 } };
```

The preceding example uses two features available in C# 9.0 and later: [or](#) [pattern combinator](#) and [record types](#).

For more information, see the [Property pattern](#) section of the feature proposal note.

Positional pattern

Beginning with C# 8.0, you use a *positional pattern* to deconstruct an expression result and match the resulting values against the corresponding nested patterns, as the following example shows:

```
public readonly struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

At the preceding example, the type of an expression contains the [Deconstruct](#) method, which is used to deconstruct an expression result. You can also match expressions of [tuple types](#) against positional patterns. In that way, you can match multiple inputs against various patterns, as the following example shows:

```

static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime visitDate)
=> (groupSize, visitDate.DayOfWeek) switch
{
    (<= 0, _) => throw new ArgumentException("Group size must be positive."),
    (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
    (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
    (>= 10, DayOfWeek.Monday) => 30.0m,
    (>= 5 and < 10, _) => 12.0m,
    (>= 10, _) => 15.0m,
    _ => 0.0m,
};

```

The preceding example uses [relational](#) and [logical](#) patterns, which are available in C# 9.0 and later.

You can use the names of tuple elements and [Deconstruct](#) parameters in a positional pattern, as the following example shows:

```

var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}"); // output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}

```

You can also extend a positional pattern in any of the following ways:

- Add a runtime type check and a variable declaration, as the following example shows:

```

public record Point2D(int X, int Y);
public record Point3D(int X, int Y, int Z);

static string PrintIfAllCoordinatesArePositive(object point) => point switch
{
    Point2D (> 0, > 0) p => p.ToString(),
    Point3D (> 0, > 0, > 0) p => p.ToString(),
    _ => string.Empty,
};

```

The preceding example uses [positional records](#) that implicitly provide the [Deconstruct](#) method.

- Use a [property pattern](#) within a positional pattern, as the following example shows:

```

public record WeightedPoint(int X, int Y)
{
    public double Weight { get; set; }
}

static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) { Weight: >= 0.0 };

```

- Combine two preceding usages, as the following example shows:

```
if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}
```

A positional pattern is a recursive pattern. That is, you can use any pattern as a nested pattern.

For more information, see the [Positional pattern](#) section of the feature proposal note.

`var` pattern

Beginning with C# 7.0, you use a `var` pattern to match any expression, including `null`, and assign its result to a new local variable, as the following example shows:

```
static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
        .Range(start: 0, count: 5)
        .Select(s => rand.Next(minValue: -10, maxValue: 11))
        .ToArray();
}
```

A `var` pattern is useful when you need a temporary variable within a Boolean expression to hold the result of intermediate calculations. You can also use a `var` pattern when you need to perform additional checks in `when` case guards of a `switch` expression or statement, as the following example shows:

```
public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2))); // output: Point { X = -1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2))); // output: Point { X = 5, Y = -2 }
}
```

In the preceding example, pattern `var (x, y)` is equivalent to a [positional pattern](#) `(var x, var y)`.

In a `var` pattern, the type of a declared variable is the compile-time type of the expression that is matched against the pattern.

For more information, see the [Var pattern](#) section of the feature proposal note.

Discard pattern

Beginning with C# 8.0, you use a *discard pattern* `_` to match any expression, including `null`, as the following example shows:

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday)); // output: 5.0
Console.WriteLine(GetDiscountInPercent(null)); // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10)); // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

In the preceding example, a discard pattern is used to handle `null` and any integer value that doesn't have the corresponding member of the `DayOfWeek` enumeration. That guarantees that a `switch` expression in the example handles all possible input values. If you don't use a discard pattern in a `switch` expression and none of the expression's patterns matches an input, the runtime [throws an exception](#). The compiler generates a warning if a `switch` expression doesn't handle all possible input values.

A discard pattern cannot be a pattern in an `is` expression or a `switch` statement. In those cases, to match any expression, use a `var` pattern with a discard: `var _`.

For more information, see the [Discard pattern](#) section of the feature proposal note.

Parenthesized pattern

Beginning with C# 9.0, you can put parentheses around any pattern. Typically, you do that to emphasize or change the precedence in [logical patterns](#), as the following example shows:

```
if (input is not (float or double))
{
    return;
}
```

C# language specification

For more information, see the following feature proposal notes:

- [Pattern matching for C# 7.0](#)
- [Recursive pattern matching \(introduced in C# 8.0\)](#)
- [Pattern-matching changes for C# 9.0](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)

+ and += operators (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `+` and `+=` operators are supported by the built-in [integral](#) and [floating-point](#) numeric types, the [string](#) type, and [delegate](#) types.

For information about the arithmetic `+` operator, see the [Unary plus and minus operators](#) and [Addition operator +](#) sections of the [Arithmetic operators](#) article.

String concatenation

When one or both operands are of type [string](#), the `+` operator concatenates the string representations of its operands (the string representation of `null` is an empty string):

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

Beginning with C# 6, [string interpolation](#) provides a more convenient way to format strings:

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

Delegate combination

For operands of the same [delegate](#) type, the `+` operator returns a new delegate instance that, when invoked, invokes the left-hand operand and then invokes the right-hand operand. If any of the operands is `null`, the `+` operator returns the value of another operand (which also might be `null`). The following example shows how delegates can be combined with the `+` operator:

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab(); // output: ab
```

To perform delegate removal, use the `-` operator.

For more information about delegate types, see [Delegates](#).

Addition assignment operator `+=`

An expression using the `+=` operator, such as

```
x += y
```

is equivalent to

```
x = x + y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of the `+=` operator:

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer(); // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer(); // output: ab
```

You also use the `+=` operator to specify an event handler method when you subscribe to an [event](#). For more information, see [How to: subscribe to and unsubscribe from events](#).

Operator overloadability

A user-defined type can [overload](#) the `+` operator. When a binary `+` operator is overloaded, the `+=` operator is also implicitly overloaded. A user-defined type cannot explicitly overload the `+=` operator.

C# language specification

For more information, see the [Unary plus operator](#) and [Addition operator](#) sections of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [How to concatenate multiple strings](#)
- [Events](#)
- [Arithmetic operators](#)
- [- and -= operators](#)

- and -= operators (C# reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The `-` and `-=` operators are supported by the built-in [integral](#) and [floating-point](#) numeric types and [delegate](#) types.

For information about the arithmetic `-` operator, see the [Unary plus and minus operators](#) and [Subtraction operator -](#) sections of the [Arithmetic operators](#) article.

Delegate removal

For operands of the same [delegate](#) type, the `-` operator returns a delegate instance that is calculated as follows:

- If both operands are non-null and the invocation list of the right-hand operand is a proper contiguous sublist of the invocation list of the left-hand operand, the result of the operation is a new invocation list that is obtained by removing the right-hand operand's entries from the invocation list of the left-hand operand. If the right-hand operand's list matches multiple contiguous sublists in the left-hand operand's list, only the right-most matching sublist is removed. If removal results in an empty list, the result is `null`.

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");

var abbaab = a + b + b + a + a + b;
abbaab(); // output: abbaab
Console.WriteLine();

var ab = a + b;
var abba = abbaab - ab;
abba(); // output: abba
Console.WriteLine();

var nihil = abbaab - abbaab;
Console.WriteLine(nihil is null); // output: True
```

- If the invocation list of the right-hand operand is not a proper contiguous sublist of the invocation list of the left-hand operand, the result of the operation is the left-hand operand. For example, removing a delegate that is not part of the multicast delegate does nothing and results in the unchanged multicast delegate.

```

Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first(); // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first)); // output: True

Action a2 = () => Console.WriteLine("a");
var changed = aba - a;
changed(); // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged(); // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged)); // output: True

```

The preceding example also demonstrates that during delegate removal delegate instances are compared. For example, delegates that are produced from evaluation of identical [lambda expressions](#) are not equal. For more information about delegate equality, see the [Delegate equality operators](#) section of the [C# language specification](#).

- If the left-hand operand is `null`, the result of the operation is `null`. If the right-hand operand is `null`, the result of the operation is the left-hand operand.

```

Action a = () => Console.WriteLine("a");

var nothing = null - a;
Console.WriteLine(nothing is null); // output: True

var first = a - null;
a(); // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a)); // output: True

```

To combine delegates, use the `+` operator.

For more information about delegate types, see [Delegates](#).

Subtraction assignment operator `-=`

An expression using the `-=` operator, such as

```
x -= y
```

is equivalent to

```
x = x - y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of the `-=` operator:

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer(); // output: aba

Console.WriteLine();
printer -= a;
printer(); // output: ab
```

You also use the `-=` operator to specify an event handler method to remove when you unsubscribe from an [event](#). For more information, see [How to subscribe to and unsubscribe from events](#).

Operator overloadability

A user-defined type can [overload](#) the `-` operator. When a binary `-` operator is overloaded, the `-=` operator is also implicitly overloaded. A user-defined type cannot explicitly overload the `-=` operator.

C# language specification

For more information, see the [Unary minus operator](#) and [Subtraction operator](#) sections of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Events](#)
- [Arithmetic operators](#)
- [+ and += operators](#)

?: operator (C# reference)

11/2/2020 • 3 minutes to read • [Edit Online](#)

The conditional operator `?:`, also known as the ternary conditional operator, evaluates a Boolean expression and returns the result of one of the two expressions, depending on whether the Boolean expression evaluates to `true` or `false`.

The syntax for the conditional operator is as follows:

```
condition ? consequent : alternative
```

The `condition` expression must evaluate to `true` or `false`. If `condition` evaluates to `true`, the `consequent` expression is evaluated, and its result becomes the result of the operation. If `condition` evaluates to `false`, the `alternative` expression is evaluated, and its result becomes the result of the operation. Only `consequent` or `alternative` is evaluated.

Beginning with C# 9.0, conditional expressions are target-typed. That is, if a target type of a conditional expression is known, the types of `consequent` and `alternative` must be implicitly convertible to the target type, as the following example shows:

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

If a target type of a conditional expression is unknown (for example, when you use the `var` keyword) or in C# 8.0 and earlier, the type of `consequent` and `alternative` must be the same or there must be an implicit conversion from one type to the other:

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

The conditional operator is right-associative, that is, an expression of the form

```
a ? b : c ? d : e
```

is evaluated as

```
a ? b : (c ? d : e)
```

TIP

You can use the following mnemonic device to remember how the conditional operator is evaluated:

```
is this condition true ? yes : no
```

The following example demonstrates the usage of the conditional operator:

```
double sinc(double x) => x != 0.0 ? Math.Sin(x) / x : 1;

Console.WriteLine(sinc(0.1));
Console.WriteLine(sinc(0.0));
// Output:
// 0.998334166468282
// 1
```

Conditional ref expression

Beginning with C# 7.2, a [ref local](#) or [ref readonly local](#) variable can be assigned conditionally with a conditional ref expression. You can also use a conditional ref expression as a [reference return value](#) or as a [ref method argument](#).

The syntax for a conditional ref expression is as follows:

```
condition ? ref consequent : ref alternative
```

Like the original conditional operator, a conditional ref expression evaluates only one of the two expressions: either `consequent` or `alternative`.

In the case of a conditional ref expression, the type of `consequent` and `alternative` must be the same. Conditional ref expressions are not target-typed.

The following example demonstrates the usage of a conditional ref expression:

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Conditional operator and an `if..else` statement

Use of the conditional operator instead of an `if..else` statement might result in more concise code in cases when you need conditionally to compute a value. The following example demonstrates two ways to classify an integer as negative or nonnegative:

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Operator overloadability

A user-defined type cannot overload the conditional operator.

C# language specification

For more information, see the [Conditional operator](#) section of the [C# language specification](#).

For more information about features added in C# 7.2 and later, see the following feature proposal notes:

- [Conditional ref expressions \(C# 7.2\)](#)
- [Target-typed conditional expression \(C# 9.0\)](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [if-else statement](#)
- [?. and ?\[\] operators](#)
- [?? and ??= operators](#)
- [ref keyword](#)

! (null-forgiving) operator (C# reference)

3/25/2021 • 2 minutes to read • [Edit Online](#)

Available in C# 8.0 and later, the unary postfix `!` operator is the null-forgiving, or null-suppression, operator. In an enabled [nullable annotation context](#), you use the null-forgiving operator to declare that expression `x` of a reference type isn't `null`: `x!`. The unary prefix `!` operator is the [logical negation operator](#).

The null-forgiving operator has no effect at run time. It only affects the compiler's static flow analysis by changing the null state of the expression. At run time, expression `x!` evaluates to the result of the underlying expression `x`.

For more information about the nullable reference types feature, see [Nullable reference types](#).

Examples

One of the use cases of the null-forgiving operator is in testing the argument validation logic. For example, consider the following class:

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Using the [MSTest test framework](#), you can create the following test for the validation logic in the constructor:

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Without the null-forgiving operator, the compiler generates the following warning for the preceding code:

`Warning CS8625: Cannot convert null literal to non-nullable reference type`. By using the null-forgiving operator, you inform the compiler that passing `null` is expected and shouldn't be warned about.

You can also use the null-forgiving operator when you definitely know that an expression cannot be `null` but the compiler doesn't manage to recognize that. In the following example, if the `IsValid` method returns `true`, its argument is not `null` and you can safely dereference it:

```

public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}

```

Without the null-forgiving operator, the compiler generates the following warning for the `p.Name` code:

`Warning CS8602: Dereference of a possibly null reference.`

If you can modify the `IsValid` method, you can use the `NotNullWhen` attribute to inform the compiler that an argument of the `IsValid` method cannot be `null` when the method returns `true`:

```

public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}

```

In the preceding example, you don't need to use the null-forgiving operator because the compiler has enough information to find out that `p` cannot be `null` inside the `if` statement. For more information about the attributes that allow you to provide additional information about the null state of a variable, see [Upgrade APIs with attributes to define null expectations](#).

C# language specification

For more information, see [The null-forgiving operator](#) section of the [draft of the nullable reference types specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Tutorial: Design with nullable reference types](#)

?? and ??= operators (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The null-coalescing operator `??` returns the value of its left-hand operand if it isn't `null`; otherwise, it evaluates the right-hand operand and returns its result. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

Available in C# 8.0 and later, the null-coalescing assignment operator `??=` assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`. The `??=` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>().Add(5);
Console.WriteLine(string.Join(" ", numbers)); // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers)); // output: 5 0
Console.WriteLine(a); // output: 0
```

The left-hand operand of the `??=` operator must be a variable, a [property](#), or an [indexer](#) element.

In C# 7.3 and earlier, the type of the left-hand operand of the `??` operator must be either a [reference type](#) or a [nullable value type](#). Beginning with C# 8.0, that requirement is replaced with the following: the type of the left-hand operand of the `??` and `??=` operators cannot be a non-nullable value type. In particular, beginning with C# 8.0, you can use the null-coalescing operators with unconstrained type parameters:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

The null-coalescing operators are right-associative. That is, expressions of the form

```
a ?? b ?? c
d ??= e ??= f
```

are evaluated as

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Examples

The `??` and `??=` operators can be useful in the following scenarios:

- In expressions with the [null-conditional operators ?.](#) and [?\[\]](#), you can use the `??` operator to provide an alternative expression to evaluate in case the result of the expression with null-conditional operations is `null`:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum); // output: NaN
```

- When you work with [nullable value types](#) and need to provide a value of an underlying value type, use the `??` operator to specify the value to provide in case a nullable type value is `null`:

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b); // output: -1
```

Use the [`Nullable<T>.GetValueOrDefault\(\)`](#) method if the value to be used when a nullable type value is `null` should be the default value of the underlying value type.

- Beginning with C# 7.0, you can use a [throw expression](#) as the right-hand operand of the `??` operator to make the argument-checking code more concise:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
}
```

The preceding example also demonstrates how to use [expression-bodied members](#) to define a property.

- Beginning with C# 8.0, you can use the `??=` operator to replace the code of the form

```
if (variable is null)
{
    variable = expression;
}
```

with the following code:

```
variable ??= expression;
```

Operator overloadability

The operators `??` and `??=` cannot be overloaded.

C# language specification

For more information about the `??` operator, see [The null coalescing operator](#) section of the [C# language specification](#).

For more information about the `??=` operator, see the [feature proposal note](#).

See also

- [C# reference](#)

- C# operators and expressions
- ?. and ?[] operators
- ?: operator

=> operator (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The => token is supported in two forms: as the [lambda operator](#) and as a separator of a member name and the member implementation in an [expression body definition](#).

Lambda operator

In [Lambda expressions](#), the lambda operator => separates the input parameters on the left side from the lambda body on the right side.

The following example uses the [LINQ](#) feature with method syntax to demonstrate the usage of lambda expressions:

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
    .Where(w => w.StartsWith("a"))
    .Min(w => w.Length);
Console.WriteLine(minimalLength); // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product); // output: 280
```

Input parameters of a lambda expression are strongly typed at compile time. When the compiler can infer the types of input parameters, like in the preceding example, you may omit type declarations. If you need to specify the type of input parameters, you must do that for each parameter, as the following example shows:

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product); // output: 280
```

The following example shows how to define a lambda expression without input parameters:

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

For more information, see [Lambda expressions](#).

Expression body definition

An expression body definition has the following general syntax:

```
member => expression;
```

where expression is a valid expression. The return type of expression must be implicitly convertible to the member's return type. If the member's return type is void or if the member is a constructor, a finalizer, or a property or indexer set accessor, expression must be a [statement expression](#). Because the expression's result is discarded, the return type of that expression can be any type.

The following example shows an expression body definition for a `Person.ToString` method:

```
public override string ToString() => $"{fname} {lname}".Trim();
```

It's a shorthand version of the following method definition:

```
public override string ToString()
{
    return $"{fname} {lname}".Trim();
}
```

Expression body definitions for methods, operators, and read-only properties are supported beginning with C# 6. Expression body definitions for constructors, finalizers, and property and indexer accessors are supported beginning with C# 7.0.

For more information, see [Expression-bodied members](#).

Operator overloadability

The `=>` operator cannot be overloaded.

C# language specification

For more information about the lambda operator, see the [Anonymous function expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)

:: operator (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Use the namespace alias qualifier `::` to access a member of an aliased namespace. You can use the `::` qualifier only between two identifiers. The left-hand identifier can be any of the following aliases:

- A namespace alias created with a [using alias directive](#):

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
point.Y);
}
```

- An [extern alias](#).
- The `global` alias, which is the global namespace alias. The global namespace is the namespace that contains namespaces and types that are not declared inside a named namespace. When used with the `::` qualifier, the `global` alias always references the global namespace, even if there is the user-defined `global` namespace alias.

The following example uses the `global` alias to access the .NET [System](#) namespace, which is a member of the global namespace. Without the `global` alias, the user-defined `System` namespace, which is a member of the `MyCompany.MyProduct` namespace, would be accessed:

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

NOTE

The `global` keyword is the global namespace alias only when it's the left-hand identifier of the `::` qualifier.

You can also use the `. token` to access a member of an aliased namespace. However, the `. token` is also used to access a type member. The `::` qualifier ensures that its left-hand identifier always references a namespace alias, even if there exists a type or namespace with the same name.

C# language specification

For more information, see the [Namespace alias qualifiers](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Using namespaces](#)

await operator (C# reference)

3/23/2021 • 3 minutes to read • [Edit Online](#)

The `await` operator suspends evaluation of the enclosing `async` method until the asynchronous operation represented by its operand completes. When the asynchronous operation completes, the `await` operator returns the result of the operation, if any. When the `await` operator is applied to the operand that represents an already completed operation, it returns the result of the operation immediately without suspension of the enclosing method. The `await` operator doesn't block the thread that evaluates the `async` method. When the `await` operator suspends the enclosing `async` method, the control returns to the caller of the method.

In the following example, the `HttpClient.GetByteArrayAsync` method returns the `Task<byte[]>` instance, which represents an asynchronous operation that produces a byte array when it completes. Until the operation completes, the `await` operator suspends the `DownloadDocs MainPageAsync` method. When `DownloadDocs MainPageAsync` gets suspended, control is returned to the `Main` method, which is the caller of `DownloadDocs MainPageAsync`. The `Main` method executes until it needs the result of the asynchronous operation performed by the `DownloadDocs MainPageAsync` method. When `GetByteArrayAsync` gets all the bytes, the rest of the `DownloadDocs MainPageAsync` method is evaluated. After that, the rest of the `Main` method is evaluated.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocs MainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocs MainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}

// Output similar to:
// DownloadDocs MainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocs MainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

The preceding example uses the `async Main` method, which is possible beginning with C# 7.1. For more information, see the [await operator in the Main method](#) section.

NOTE

For an introduction to asynchronous programming, see [Asynchronous programming with `async` and `await`](#). Asynchronous programming with `async` and `await` follows the [task-based asynchronous pattern](#).

You can use the `await` operator only in a method, [lambda expression](#), or [anonymous method](#) that is modified by the `async` keyword. Within an `async` method, you can't use the `await` operator in the body of a synchronous function, inside the block of a [lock statement](#), and in an [unsafe](#) context.

The operand of the `await` operator is usually of one of the following .NET types: `Task`, `Task<TResult>`, `ValueTask`, or `ValueTask<TResult>`. However, any awaitable expression can be the operand of the `await` operator. For more information, see the [Awaitable expressions](#) section of the [C# language specification](#).

The type of expression `await t` is `TResult` if the type of expression `t` is `Task<TResult>` or `ValueTask<TResult>`. If the type of `t` is `Task` or `ValueTask`, the type of `await t` is `void`. In both cases, if `t` throws an exception, `await t` rethrows the exception. For more information about exception handling, see the [Exceptions in `async` methods](#) section of the [try-catch statement](#) article.

The `async` and `await` keywords are available in C# 5 and later.

Asynchronous streams and disposables

Beginning with C# 8.0, you can work with asynchronous streams and disposables.

You use the `await foreach` statement to consume an asynchronous stream of data. For more information, see the [foreach statement](#) article and the [Asynchronous streams](#) section of the [What's new in C# 8.0](#) article.

You use the `await using` statement to work with an asynchronously disposable object, that is, an object of a type that implements an [IAsyncDisposable](#) interface. For more information, see the [Using `async` disposable](#) section of the [Implement a `DisposeAsync` method](#) article.

await operator in the Main method

Beginning with C# 7.1, the `Main` method, which is the application entry point, can return `Task` or `Task<int>`, enabling it to be `async` so you can use the `await` operator in its body. In earlier C# versions, to ensure that the `Main` method waits for the completion of an asynchronous operation, you can retrieve the value of the `Task<TResult>.Result` property of the `Task<TResult>` instance that is returned by the corresponding `async` method. For asynchronous operations that don't produce a value, you can call the `Task.Wait` method. For information about how to select the language version, see [C# language versioning](#).

C# language specification

For more information, see the [Await expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [async](#)
- [Task asynchronous programming model](#)
- [Asynchronous programming](#)
- [Async in depth](#)
- [Walkthrough: accessing the Web by using `async` and `await`](#)

- [Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0](#)

default value expressions (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A default value expression produces the [default value](#) of a type. There are two kinds of default value expressions: the [default operator](#) call and a [default literal](#).

You also use the `default` keyword as the default case label within a [switch statement](#).

default operator

The argument to the `default` operator must be the name of a type or a type parameter, as the following example shows:

```
Console.WriteLine(default(int)); // output: 0
Console.WriteLine(default(object) is null); // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

default literal

Beginning with C# 7.1, you can use the `default` literal to produce the default value of a type when the compiler can infer the expression type. The `default` literal expression produces the same value as the `default(T)` expression where `T` is the inferred type. You can use the `default` literal in any of the following cases:

- In the assignment or initialization of a variable.
- In the declaration of the default value for an [optional method parameter](#).
- In a method call to provide an argument value.
- In a [return statement](#) or as an expression in an [expression-bodied member](#).

The following example shows the usage of the `default` literal:

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3)); // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default)); // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue)); // output: [ (0, 0), (0, 0), (0, 0) ]
```

C# language specification

For more information, see the [Default value expressions](#) section of the [C# language specification](#).

For more information about the `default` literal, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Default values of C# types](#)
- [Generics in .NET](#)

delegate operator (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `delegate` operator creates an anonymous method that can be converted to a delegate type:

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4)); // output: 7
```

NOTE

Beginning with C# 3, lambda expressions provide a more concise and expressive way to create an anonymous function. Use the `=>` operator to construct a lambda expression:

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4)); // output: 7
```

For more information about features of lambda expressions, for example, capturing outer variables, see [Lambda expressions](#).

When you use the `delegate` operator, you might omit the parameter list. If you do that, the created anonymous method can be converted to a delegate type with any list of parameters, as the following example shows:

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

That's the only functionality of anonymous methods that is not supported by lambda expressions. In all other cases, a lambda expression is a preferred way to write inline code.

Beginning with C# 9.0, you can use `discards` to specify two or more input parameters of an anonymous method that aren't used by the method:

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4)); // output: 42
```

For backwards compatibility, if only a single parameter is named `__`, `__` is treated as the name of that parameter within an anonymous method.

Also beginning with C# 9.0, you can use the `static` modifier at the declaration of an anonymous method:

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4)); // output: 14
```

A static anonymous method can't capture local variables or instance state from enclosing scopes.

You also use the `delegate` keyword to declare a [delegate type](#).

C# language specification

For more information, see the [Anonymous function expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [=> operator](#)

nameof expression (C# reference)

3/6/2021 • 2 minutes to read • [Edit Online](#)

A `nameof` expression produces the name of a variable, type, or member as the string constant:

```
Console.WriteLine(nameof(System.Collections.Generic)); // output: Generic
Console.WriteLine(nameof(List<int>)); // output: List
Console.WriteLine(nameof(List<int>.Count)); // output: Count
Console.WriteLine(nameof(List<int>.Add)); // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers)); // output: numbers
Console.WriteLine(nameof(numbers.Count)); // output: Count
Console.WriteLine(nameof(numbers.Add)); // output: Add
```

As the preceding example shows, in the case of a type and a namespace, the produced name is not [fully qualified](#).

In the case of [verbatim identifiers](#), the `@` character is not the part of a name, as the following example shows:

```
var @new = 5;
Console.WriteLine(nameof(@new)); // output: new
```

A `nameof` expression is evaluated at compile time and has no effect at run time.

You can use a `nameof` expression to make the argument-checking code more maintainable:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

A `nameof` expression is available in C# 6 and later.

C# language specification

For more information, see the [Nameof expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)

new operator (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `new` operator creates a new instance of a type.

You can also use the `new` keyword as a [member declaration modifier](#) or a [generic type constraint](#).

Constructor invocation

To create a new instance of a type, you typically invoke one of the [constructors](#) of that type using the `new` operator:

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

You can use an [object or collection initializer](#) with the `new` operator to instantiate and initialize an object in one statement, as the following example shows:

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Beginning with C# 9.0, constructor invocation expressions are target-typed. That is, if a target type of an expression is known, you can omit a type name, as the following example shows:

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

As the preceding example shows, you always use parentheses in a target-typed `new` expression.

If a target type of a `new` expression is unknown (for example, when you use the `var` keyword), you must specify a type name.

Array creation

You also use the `new` operator to create an array instance, as the following example shows:

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Use array initialization syntax to create an array instance and populate it with elements in one statement. The following example shows various ways how you can do that:

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType()); // output: System.Int32[]
```

For more information about arrays, see [Arrays](#).

Instantiation of anonymous types

To create an instance of an [anonymous type](#), use the `new` operator and object initializer syntax:

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destruction of type instances

You don't have to destroy earlier created type instances. Instances of both reference and value types are destroyed automatically. Instances of value types are destroyed as soon as the context that contains them is destroyed. Instances of reference types are destroyed by the [garbage collector](#) at some unspecified time after the last reference to them is removed.

For type instances that contain unmanaged resources, for example, a file handle, it's recommended to employ deterministic clean-up to ensure that the resources they contain are released as soon as possible. For more information, see the [System.IDisposable](#) API reference and the [using statement](#) article.

Operator overloadability

A user-defined type cannot overload the `new` operator.

C# language specification

For more information, see [The new operator](#) section of the [C# language specification](#).

For more information about a target-typed `new` expression, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Object and collection initializers](#)

sizeof operator (C# reference)

4/6/2021 • 2 minutes to read • [Edit Online](#)

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The argument to the `sizeof` operator must be the name of an [unmanaged type](#) or a type parameter that is [constrained](#) to be an unmanaged type.

The `sizeof` operator requires an [unsafe](#) context. However, the expressions presented in the following table are evaluated in compile time to the corresponding constant values and don't require an unsafe context:

EXPRESSION	CONSTANT VALUE
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(decimal)</code>	16
<code>sizeof(bool)</code>	1

You also don't need to use an unsafe context when the operand of the `sizeof` operator is the name of an [enum](#) type.

The following example demonstrates the usage of the `sizeof` operator:

```

using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte)); // output: 1
        Console.WriteLine(sizeof(double)); // output: 8

        DisplaySizeOf<Point>(); // output: Size of Point is 24
        DisplaySizeOf<decimal>(); // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*)); // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}

```

The `sizeof` operator returns a number of bytes that would be allocated by the common language runtime in managed memory. For `struct` types, that value includes any padding, as the preceding example demonstrates. The result of the `sizeof` operator might differ from the result of the `Marshal.SizeOf` method, which returns the size of a type in *unmanaged* memory.

C# language specification

For more information, see [The sizeof operator](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Pointer related operators](#)
- [Pointer types](#)
- [Memory and span-related types](#)
- [Generics in .NET](#)

stackalloc expression (C# reference)

4/6/2021 • 3 minutes to read • [Edit Online](#)

A `stackalloc` expression allocates a block of memory on the stack. A stack allocated memory block created during the method execution is automatically discarded when that method returns. You cannot explicitly free the memory allocated with `stackalloc`. A stack allocated memory block is not subject to [garbage collection](#) and doesn't have to be pinned with a `fixed` statement.

You can assign the result of a `stackalloc` expression to a variable of one of the following types:

- Beginning with C# 7.2, `System.Span<T>` or `System.ReadOnlySpan<T>`, as the following example shows:

```
int length = 3;
Span<int> numbers = stackalloc int[length];
for (var i = 0; i < length; i++)
{
    numbers[i] = i;
}
```

You don't have to use an [unsafe](#) context when you assign a stack allocated memory block to a `Span<T>` or `ReadOnlySpan<T>` variable.

When you work with those types, you can use a `stackalloc` expression in [conditional](#) or assignment expressions, as the following example shows:

```
int length = 1000;
Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
```

Beginning with C# 8.0, you can use a `stackalloc` expression inside other expressions whenever a `Span<T>` or `ReadOnlySpan<T>` variable is allowed, as the following example shows:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

NOTE

We recommend using `Span<T>` or `ReadOnlySpan<T>` types to work with stack allocated memory whenever possible.

- A [pointer type](#), as the following example shows:

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

As the preceding example shows, you must use an `unsafe` context when you work with pointer types.

In the case of pointer types, you can use a `stackalloc` expression only in a local variable declaration to initialize the variable.

The amount of memory available on the stack is limited. If you allocate too much memory on the stack, a [StackOverflowException](#) is thrown. To avoid that, follow the rules below:

- Limit the amount of memory you allocate with `stackalloc`:

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
byte[inputLength];
```

Because the amount of memory available on the stack depends on the environment in which the code is executed, be conservative when you define the actual limit value.

- Avoid using `stackalloc` inside loops. Allocate the memory block outside a loop and reuse it inside the loop.

The content of the newly allocated memory is undefined. You should initialize it before the use. For example, you can use the [Span<T>.Clear](#) method that sets all the items to the default value of type `T`.

Beginning with C# 7.3, you can use array initializer syntax to define the content of the newly allocated memory. The following example demonstrates various ways to do that:

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

In expression `stackalloc T[E]`, `T` must be an [unmanaged type](#) and `E` must evaluate to a non-negative `int` value.

Security

The use of `stackalloc` automatically enables buffer overrun detection features in the common language runtime (CLR). If a buffer overrun is detected, the process is terminated as quickly as possible to minimize the chance that malicious code is executed.

C# language specification

For more information, see the [Stack allocation](#) section of the [C# language specification](#) and the [Permit stackalloc in nested contexts](#) feature proposal note.

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Pointer related operators](#)
- [Pointer types](#)
- [Memory and span-related types](#)
- [Dos and Don'ts of stackalloc](#)

switch expression (C# reference)

3/6/2021 • 3 minutes to read • [Edit Online](#)

This article covers the `switch` expression, introduced in C# 8.0. For information on the `switch` statement, see the article on the `switch` statement in the [statements](#) section.

Basic example

The `switch` expression provides for `switch`-like semantics in an expression context. It provides a concise syntax when the switch arms produce a value. The following example shows the structure of a switch expression. It translates values from an `enum` representing visual directions in an online map to the corresponding cardinal direction:

```
public static class SwitchExample
{
    public enum Directions
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static void Main()
    {
        var direction = Directions.Right;
        Console.WriteLine($"Map view direction is {direction}");

        var orientation = direction switch
        {
            Directions.Up      => Orientation.North,
            Directions.Right   => Orientation.East,
            Directions.Down    => Orientation.South,
            Directions.Left    => Orientation.West,
        };
        Console.WriteLine($"Cardinal orientation is {orientation}");
    }
}
```

The preceding sample shows the basic elements of a switch expression:

- The *range expression*: The preceding example uses the variable `direction` as the range expression.
- The *switch expression arms*: Each switch expression arm contains a *pattern*, an optional *case guard*, the `=>` token, and an *expression*.

The result of the `switch expression` is the value of the expression of the first `switch expression arm` whose *pattern* matches the *range expression* and whose *case guard*, if present, evaluates to `true`. The *expression* on the right of the `=>` token can't be an expression statement.

The *switch expression arms* are evaluated in text order. The compiler issues an error when a lower *switch expression arm* can't be chosen because a higher *switch expression arm* matches all its values.

Patterns and case guards

Many patterns are supported in switch expression arms. The preceding example uses a *constant pattern*. A *constant pattern* compares the range expression to a value. That value must be a compile-time constant. The *type pattern* compares the range expression to a known type. The following example retrieves the third element from a sequence. It uses different methods based on the type of the sequence:

```
public static T TypeExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array array => (T)array.GetValue(2),
        IList<T> list      => list[2],
        IEnumerable<T> seq   => seq.Skip(2).First(),
    };
```

Patterns can be recursive, where a pattern tests a type, and if that type matches, the pattern matches one or more property values on the range expression. You can use recursive patterns to extend the preceding example. You add switch expression arms for arrays that have fewer than 3 elements. Recursive patterns are shown in the following example:

```
public static T RecursiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}      => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array             => (T)array.GetValue(2),
        IList<T> list                => list[2],
        IEnumerable<T> seq            => seq.Skip(2).First(),
    };
```

Recursive patterns can examine properties of the range expression, but can't execute arbitrary code. You can use a *case guard*, specified in a `when` clause, to provide similar checks for other sequence types:

```
public static T CaseGuardExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}      => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array             => (T)array.GetValue(2),
        IEnumerable<T> list when !list.Any() => default(T),
        IEnumerable<T> list when list.Count() < 3 => list.Last(),
        IList<T> list                 => list[2],
        IEnumerable<T> seq            => seq.Skip(2).First(),
    };
```

Finally, you can add the `_` pattern and the `null` pattern to catch arguments that aren't processed by any other switch expression arm. That makes the switch expression *exhaustive*, meaning any possible value of the range expression is handled. The following example adds those expression arms:

```

public static T ExhaustiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}      => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array            => (T)array.GetValue(2),
        IEnumerable<T> list
            when !list.Any()          => default(T),
        IEnumerable<T> list
            when list.Count() < 3     => list.Last(),
        IList<T> list                => list[2],
        null                         => throw new ArgumentNullException(nameof(sequence)),
        _                            => sequence.Skip(2).First(),
    };

```

The preceding example adds a `null` pattern, and changes the `IEnumerable<T>` type pattern to a `_` pattern. The `null` pattern provides a null check as a switch expression arm. The expression for that arm throws an [ArgumentNullException](#). The `_` pattern matches all inputs that haven't been matched by previous arms. It must come after the `null` check, or it would match `null` inputs.

Non-exhaustive switch expressions

If none of a switch expression's patterns catches an argument, the runtime throws an exception. In .NET Core 3.0 and later versions, the exception is a [System.Runtime.CompilerServices.SwitchExpressionException](#). In .NET Framework, the exception is an [InvalidOperationException](#).

See also

- [C# language spec proposal for recursive patterns](#)
- [C# reference](#)
- [C# operators and expressions](#)
- [Pattern matching](#)

true and false operators (C# reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `true` operator returns the `bool` value `true` to indicate that its operand is definitely true. The `false` operator returns the `bool` value `true` to indicate that its operand is definitely false. The `true` and `false` operators are not guaranteed to complement each other. That is, both the `true` and `false` operator might return the `bool` value `false` for the same operand. If a type defines one of the two operators, it must also define another operator.

TIP

Use the `bool?` type, if you need to support the three-valued logic (for example, when you work with databases that support a three-valued Boolean type). C# provides the `&` and `|` operators that support the three-valued logic with the `bool?` operands. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

Boolean expressions

A type with the defined `true` operator can be the type of a result of a controlling conditional expression in the `if`, `do`, `while`, and `for` statements and in the `conditional operator` `?:`. For more information, see the [Boolean expressions](#) section of the [C# language specification](#).

User-defined conditional logical operators

If a type with the defined `true` and `false` operators [overloads](#) the `logical OR operator` `|` or the `logical AND operator` `&` in a certain way, the `conditional logical OR operator` `||` or `conditional logical AND operator` `&&`, respectively, can be evaluated for the operands of that type. For more information, see the [User-defined conditional logical operators](#) section of the [C# language specification](#).

Example

The following example presents the type that defines both `true` and `false` operators. The type also overloads the `logical AND operator` `&` in such a way that the `&&` operator also can be evaluated for the operands of that type.

```

using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}

```

Notice the short-circuiting behavior of the `&&` operator. When the `GetFuelLaunchStatus` method returns `LaunchStatus.Red`, the right-hand operand of the `&&` operator is not evaluated. That is because `LaunchStatus.Red` is definitely false. Then the result of the logical AND doesn't depend on the value of the right-hand operand. The output of the example is as follows:

```
Getting fuel launch status...
Wait!
```

See also

- [C# reference](#)
- [C# operators and expressions](#)

with expression (C# reference)

3/23/2021 • 3 minutes to read • [Edit Online](#)

Available in C# 9.0 and later, a `with` expression produces a copy of its `record` operand with the specified properties and fields modified:

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}"); // output: p2: NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
        {
            Name = "C",
            Y = 4
        };
        Console.WriteLine($"{nameof(p3)}: {p3}"); // output: p3: NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }
    }
}
```

As the preceding example shows, you use `object initializer` syntax to specify what members to modify and their new values. In a `with` expression, a left-hand operand must be of a record type.

The result of a `with` expression has the same runtime type as the expression's operand, as the following example shows:

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint); // output: True
        Console.WriteLine(p2); // output: NamedPoint { X = 5, Y = 3, Name = A }

    }
}
```

In the case of a reference-type member, only the reference to an instance is copied when a record is copied. Both the copy and original record have access to the same reference-type instance. The following example demonstrates that behavior:

```

using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}

```

Any record type has the *copy constructor*. That is a constructor with a single parameter of the containing record type. It copies the state of its argument to a new record instance. At evaluation of a `with` expression, the copy constructor gets called to instantiate a new record instance based on an original record. After that, the new instance gets updated according to the specified modifications. By default, the copy constructor is implicit, that is, compiler-generated. If you need to customize the record copy semantics, explicitly declare a copy constructor with the desired behavior. The following example updates the preceding example with an explicit copy constructor. The new copy behavior is to copy list items instead of a list reference when a record is copied:

```

using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}

```

C# language specification

For more information, see the following sections of the [records feature proposal note](#):

- [with expression](#)
- [Copy and Clone members](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Records](#)

Operator overloading (C# reference)

3/6/2021 • 3 minutes to read • [Edit Online](#)

A user-defined type can overload a predefined C# operator. That is, a type can provide the custom implementation of an operation in case one or both of the operands are of that type. The [Overloadable operators](#) section shows which C# operators can be overloaded.

Use the `operator` keyword to declare an operator. An operator declaration must satisfy the following rules:

- It includes both a `public` and a `static` modifier.
- A unary operator has one input parameter. A binary operator has two input parameters. In each case, at least one parameter must have type `T` or `T?` where `T` is the type that contains the operator declaration.

The following example defines a simplified structure to represent a rational number. The structure overloads some of the [arithmetic operators](#):

```

using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a); // output: -5 / 4
        Console.WriteLine(a + b); // output: 14 / 8
        Console.WriteLine(a - b); // output: 6 / 8
        Console.WriteLine(a * b); // output: 5 / 8
        Console.WriteLine(a / b); // output: 10 / 4
    }
}

```

You could extend the preceding example by [defining an implicit conversion](#) from `int` to `Fraction`. Then, overloaded operators would support arguments of those two types. That is, it would become possible to add an integer to a fraction and obtain a fraction as a result.

You also use the `operator` keyword to define a custom type conversion. For more information, see [User-defined conversion operators](#).

Overloadable operators

The following table provides information about overloadability of C# operators:

OPERATORS	OVERLOADABILITY
<code>+x, -x, !x, ~x, ++, --, true, false</code>	These unary operators can be overloaded.
<code>x + y, x - y, x * y, x / y, x % y, x & y, x y, x ^ y, x << y, x >> y, x == y, x != y, x < y, x > y, x <= y, x >= y</code>	These binary operators can be overloaded. Certain operators must be overloaded in pairs; for more information, see the note that follows this table.
<code>x && y, x y</code>	Conditional logical operators cannot be overloaded. However, if a type with the overloaded <code>true</code> and <code>false</code> operators also overloads the <code>&</code> or <code> </code> operator in a certain way, the <code>&&</code> or <code> </code> operator, respectively, can be evaluated for the operands of that type. For more information, see the User-defined conditional logical operators section of the C# language specification .
<code>a[i], a?[i]</code>	Element access is not considered an overloadable operator, but you can define an indexer .
<code>(T)x</code>	The cast operator cannot be overloaded, but you can define custom type conversions that can be performed by a cast expression. For more information, see User-defined conversion operators .
<code>+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=</code>	Compound assignment operators cannot be explicitly overloaded. However, when you overload a binary operator, the corresponding compound assignment operator, if any, is also implicitly overloaded. For example, <code>+=</code> is evaluated using <code>+</code> , which can be overloaded.
<code>^x, x = y, x.y, x?.y, c ? t : f, x ?? y, x ??= y, x.y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with</code>	These operators cannot be overloaded.

NOTE

The comparison operators must be overloaded in pairs. That is, if either operator of a pair is overloaded, the other operator must be overloaded as well. Such pairs are as follows:

- `==` and `!=` operators
- `<` and `>` operators
- `<=` and `>=` operators

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Operator overloading](#)
- [Operators](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)

- User-defined conversion operators
- Design guidelines - Operator overloads
- Design guidelines - Equality operators
- Why are overloaded operators always static in C#?

C# Special Characters

11/2/2020 • 2 minutes to read • [Edit Online](#)

Special characters are predefined, contextual characters that modify the program element (a literal string, an identifier, or an attribute name) to which they are prepended. C# supports the following special characters:

- `@`, the verbatim identifier character.
- `$`, the interpolated string character.

See also

- [C# Reference](#)
- [C# Programming Guide](#)

\$ - string interpolation (C# reference)

4/21/2020 • 4 minutes to read • [Edit Online](#)

The `$` special character identifies a string literal as an *interpolated string*. An interpolated string is a string literal that might contain *interpolation expressions*. When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results. This feature is available starting with C# 6.

String interpolation provides a more readable and convenient syntax to create formatted strings than a [string composite formatting](#) feature. The following example uses both features to produce the same output:

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

Structure of an interpolated string

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You cannot have any white space between the `$` and the `"` that starts a string literal.

The structure of an item with an interpolation expression is as follows:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Elements in square brackets are optional. The following table describes each element:

ELEMENT	DESCRIPTION
<code>interpolationExpression</code>	The expression that produces a result to be formatted. String representation of <code>null</code> is String.Empty .
<code>alignment</code>	The constant expression whose value defines the minimum number of characters in the string representation of the expression result. If positive, the string representation is right-aligned; if negative, it's left-aligned. For more information, see Alignment Component .
<code>formatString</code>	A format string that is supported by the type of the expression result. For more information, see Format String Component .

The following example uses optional formatting components described above:

```

Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi
number");
// Expected output is:
// |Left    | Right|
//      3.14159265358979 - default formatting of the pi number
//                      3.142 - display only three decimal digits of the pi number

```

Special characters

To include a brace, "{" or "}", in the text produced by an interpolated string, use two braces, "{{" or "}}". For more information, see [Escaping Braces](#).

As the colon (":") has special meaning in an interpolation expression item, in order to use a [conditional operator](#) in an interpolation expression, enclose that expression in parentheses.

The following example shows how to include a brace in a result string and how to use a conditional operator in an interpolation expression:

```

string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :{-{{
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-
// Horace is 34 years old.

```

An interpolated verbatim string starts with the \$ character followed by the @ character. For more information about verbatim strings, see the [string](#) and [verbatim identifier](#) topics.

NOTE

Starting with C# 8.0, you can use the \$ and @ tokens in any order: both \${@"..." } and {@\$"..." } are valid interpolated verbatim strings. In earlier C# versions, the \$ token must appear before the @ token.

Implicit conversions and how to specify [IFormatProvider](#) implementation

There are three implicit conversions from an interpolated string:

1. Conversion of an interpolated string to a [String](#) instance that is the result of interpolated string resolution with interpolation expression items being replaced with the properly formatted string representations of their results. This conversion uses the [CurrentCulture](#) to format expression results.
2. Conversion of an interpolated string to a [FormattableString](#) instance that represents a composite format string along with the expression results to be formatted. That allows you to create multiple result strings with culture-specific content from a single [FormattableString](#) instance. To do that, call one of the following methods:
 - A [ToString\(\)](#) overload that produces a result string for the [CurrentCulture](#).
 - An [Invariant](#) method that produces a result string for the [InvariantCulture](#).
 - A [ToString\(IFormatProvider\)](#) method that produces a result string for a specified culture.

You can also use the [ToString\(IFormatProvider\)](#) method to provide a user-defined implementation of the [IFormatProvider](#) interface that supports custom formatting. For more information, see the [Custom formatting with ICustomFormatter](#) section of the [Formatting types in .NET](#) article.

- Conversion of an interpolated string to an [IFormattable](#) instance that also allows you to create multiple result strings with culture-specific content from a single [IFormattable](#) instance.

The following example uses implicit conversion to [FormattableString](#) to create culture-specific result strings:

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");

// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Additional resources

If you are new to string interpolation, see the [String interpolation in C#](#) interactive tutorial. You can also check another [String interpolation in C#](#) tutorial that demonstrates how to use interpolated strings to produce formatted strings.

Compilation of interpolated strings

If an interpolated string has the type `string`, it's typically transformed into a [String.Format](#) method call. The compiler may replace [String.Format](#) with [String.Concat](#) if the analyzed behavior would be equivalent to concatenation.

If an interpolated string has the type [IFormattable](#) or [FormattableString](#), the compiler generates a call to the [FormattableStringFactory.Create](#) method.

C# language specification

For more information, see the [Interpolated strings](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# special characters](#)
- [Strings](#)
- [Standard numeric format strings](#)
- [Composite formatting](#)
- [String.Format](#)

@ (C# Reference)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `@` special character serves as a verbatim identifier. It can be used in the following ways:

1. To enable C# keywords to be used as identifiers. The `@` character prefixes a code element that the compiler is to interpret as an identifier rather than a C# keyword. The following example uses the `@` character to define an identifier named `for` that it uses in a `for` loop.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
    Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

2. To indicate that a string literal is to be interpreted verbatim. The `@` character in this instance defines a *verbatim string literal*. Simple escape sequences (such as `"\\\"` for a backslash), hexadecimal escape sequences (such as `"\x0041"` for an uppercase A), and Unicode escape sequences (such as `"\u0041"` for an uppercase A) are interpreted literally. Only a quote escape sequence (`"\"`) is not interpreted literally; it produces one double quotation mark. Additionally, in case of a verbatim [interpolated string](#) brace escape sequences (`{}{}` and `{}{}}`) are not interpreted literally; they produce single brace characters. The following example defines two identical file paths, one by using a regular string literal and the other by using a verbatim string literal. This is one of the more common uses of verbatim string literals.

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt
```

The following example illustrates the effect of defining a regular string literal and a verbatim string literal that contain identical character sequences.

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//      He said, "This is the last chance!"
//      He said, "This is the last \u0063hance\x0021"
```

3. To enable the compiler to distinguish between attributes in cases of a naming conflict. An attribute is a class that derives from [Attribute](#). Its type name typically includes the suffix **Attribute**, although the compiler does not enforce this convention. The attribute can then be referenced in code either by its full

type name (for example, `[InfoAttribute]` or its shortened name (for example, `[Info]`)). However, a naming conflict occurs if two shortened attribute type names are identical, and one type name includes the **Attribute** suffix but the other does not. For example, the following code fails to compile because the compiler cannot determine whether the `Info` or `InfoAttribute` attribute is applied to the `Example` class. See [CS1614](#) for more information.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
    private string information;

    public Info(string info)
    {
        information = info;
    }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
    private string information;

    public InfoAttribute(string info)
    {
        information = info;
    }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info'. Specify the full name 'InfoAttribute' to select it.
public class Example
{
    [InfoAttribute("The entry point.")]
    public static void Main()
    {
    }
}
```

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Special Characters](#)

Reserved attributes: Assembly level attributes

4/15/2020 • 2 minutes to read • [Edit Online](#)

Most attributes are applied to specific language elements such as classes or methods; however, some attributes are global—they apply to an entire assembly or module. For example, the [AssemblyVersionAttribute](#) attribute can be used to embed version information into an assembly, like this:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Global attributes appear in the source code after any top level `using` directives and before any type, module, or namespace declarations. Global attributes can appear in multiple source files, but the files must be compiled in a single compilation pass. Visual Studio adds global attributes to the `AssemblyInfo.cs` file in .NET Framework projects. These attributes aren't added to .NET Core projects.

Assembly attributes are values that provide information about an assembly. They fall into the following categories:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes

Assembly identity attributes

Three attributes (with a strong name, if applicable) determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when you reference it in code. You can set an assembly's version and culture using attributes. However, the name value is set by the compiler, the Visual Studio IDE in the [Assembly Information Dialog Box](#), or the Assembly Linker (Al.exe) when the assembly is created. The assembly name is based on the assembly manifest. The [AssemblyFlagsAttribute](#) attribute specifies whether multiple copies of the assembly can coexist.

The following table shows the identity attributes.

ATTRIBUTE	PURPOSE
AssemblyVersionAttribute	Specifies the version of an assembly.
AssemblyCultureAttribute	Specifies which culture the assembly supports.
AssemblyFlagsAttribute	Specifies whether an assembly supports side-by-side execution on the same computer, in the same process, or in the same application domain.

Informational attributes

You use informational attributes to provide additional company or product information for an assembly. The following table shows the informational attributes defined in the [System.Reflection](#) namespace.

ATTRIBUTE	PURPOSE
AssemblyProductAttribute	Specifies a product name for an assembly manifest.
AssemblyTrademarkAttribute	Specifies a trademark for an assembly manifest.
AssemblyInformationalVersionAttribute	Specifies an informational version for an assembly manifest.
AssemblyCompanyAttribute	Specifies a company name for an assembly manifest.
AssemblyCopyrightAttribute	Defines a custom attribute that specifies a copyright for an assembly manifest.
AssemblyFileVersionAttribute	Sets a specific version number for the Win32 file version resource.
CLSCompliantAttribute	Indicates whether the assembly is compliant with the Common Language Specification (CLS).

Assembly manifest attributes

You can use assembly manifest attributes to provide information in the assembly manifest. The attributes include title, description, default alias, and configuration. The following table shows the assembly manifest attributes defined in the [System.Reflection](#) namespace.

ATTRIBUTE	PURPOSE
AssemblyTitleAttribute	Specifies an assembly title for an assembly manifest.
AssemblyDescriptionAttribute	Specifies an assembly description for an assembly manifest.
AssemblyConfigurationAttribute	Specifies an assembly configuration (such as retail or debug) for an assembly manifest.
AssemblyDefaultAliasAttribute	Defines a friendly default alias for an assembly manifest.

Reserved attributes: Determine caller information

4/15/2020 • 2 minutes to read • [Edit Online](#)

Using info attributes, you obtain information about the caller to a method. You obtain the file path of the source code, the line number in the source code, and the member name of the caller. To obtain member caller information, you use attributes that are applied to optional parameters. Each optional parameter specifies a default value. The following table lists the Caller Info attributes that are defined in the [System.Runtime.CompilerServices](#) namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePathAttribute	Full path of the source file that contains the caller. The full path is the path at compile time.	String
CallerLineNumberAttribute	Line number in the source file from which the method is called.	Integer
CallerMemberNameAttribute	Method name or property name of the caller.	String

This information helps you write tracing, debugging, and create diagnostic tools. The following example shows how to use caller info attributes. On each call to the `TraceMessage` method, the caller information is substituted as arguments to the optional parameters.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
// message: Something happened.
// member name: DoProcessing
// source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
// source line number: 31
```

You specify an explicit default value for each optional parameter. You can't apply caller info attributes to parameters that aren't specified as optional. The caller info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted. Caller info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the `StackTrace` property for exceptions, the results aren't affected by obfuscation. You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the [INotifyPropertyChanged](#) interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

CALLS OCCUR WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the method or property to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See also

- [Named and Optional Arguments](#)
- [System.Reflection](#)
- [Attribute](#)
- [Attributes](#)

Reserved attributes contribute to the compiler's null state static analysis

3/18/2021 • 14 minutes to read • [Edit Online](#)

In a nullable context, the compiler performs static analysis of code to determine the null state of all reference type variables:

- *not null*: Static analysis determines that a variable is assigned a non-null value.
- *maybe null*: Static analysis can't determine that a variable is assigned a non-null value.

You can apply attributes that provide information to the compiler about the semantics of your APIs. That information helps the compiler perform static analysis and determine when a variable isn't null. This article provides a brief description of each of those attributes and how to use them. All the examples assume C# 8.0 or newer, and the code is in a nullable context.

Let's start with a familiar example. Imagine your library has the following API to retrieve a resource string:

```
bool TryGetMessage(string key, out string message)
```

The preceding example follows the familiar `Try*` pattern in .NET. There are two reference arguments for this API: the `key` and the `message` parameter. This API has the following rules relating to the nullness of these arguments:

- Callers shouldn't pass `null` as the argument for `key`.
- Callers can pass a variable whose value is `null` as the argument for `message`.
- If the `TryGetMessage` method returns `true`, the value of `message` isn't null. If the return value is `false`, the value of `message` (and its null state) is null.

The rule for `key` can be expressed by the variable type: `key` should be a non-nullable reference type. The `message` parameter is more complex. It allows `null` as the argument, but guarantees that, on success, that `out` argument isn't null. For these scenarios, you need a richer vocabulary to describe the expectations.

Several attributes have been added to express additional information about the null state of variables. All code you wrote before C# 8 introduced nullable reference types was *null oblivious*. That means any reference type variable may be null, but null checks aren't required. Once your code is *nullable aware*, those rules change. Reference types should never be the `null` value, and nullable reference types must be checked against `null` before being dereferenced.

The rules for your APIs are likely more complicated, as you saw with the `TryGetValue` API scenario. Many of your APIs have more complex rules for when variables can or can't be `null`. In these cases, you'll use one of the following attributes to express those rules:

- `AllowNull`: A non-nullable argument may be null.
- `DisallowNull`: A nullable argument should never be null.
- `MaybeNull`: A non-nullable return value may be null.
- `NotNull`: A nullable return value will never be null.
- `MaybeNullWhen`: A non-nullable argument may be null when the method returns the specified `bool` value.
- `NotNullWhen`: A nullable argument won't be null when the method returns the specified `bool` value.
- `NotNullIfNotNull`: A return value isn't null if the argument for the specified parameter isn't null.

- [DoesNotReturn](#): A method never returns. In other words, it always throws an exception.
- [DoesNotReturnIf](#): This method never returns if the associated `bool` parameter has the specified value.
- [MemberNotNull](#): The listed member won't be null when the method returns.
- [MemberNotNullWhen](#): The listed member won't be null when the method returns the specified `bool` value.

The preceding descriptions are a quick reference to what each attribute does. Each following section describes the behavior and meaning more thoroughly.

Adding these attributes gives the compiler more information about the rules for your API. When calling code is compiled in a nullable enabled context, the compiler will warn callers when they violate those rules. These attributes don't enable more checks on your implementation.

Specify preconditions: `AllowNull` and `DisallowNull`

Consider a read/write property that never returns `null` because it has a reasonable default value. Callers pass `null` to the set accessor when setting it to that default value. For example, consider a messaging system that asks for a screen name in a chat room. If none is provided, the system generates a random name:

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

When you compile the preceding code in a nullable oblivious context, everything is fine. Once you enable nullable reference types, the `ScreenName` property becomes a non-nullable reference. That's correct for the `get` accessor: it never returns `null`. Callers don't need to check the returned property for `null`. But now setting the property to `null` generates a warning. To support this type of code, you add the [System.Diagnostics.CodeAnalysis.AllowNullAttribute](#) attribute to the property, as shown in the following code:

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

You may need to add a `using` directive for [System.Diagnostics.CodeAnalysis](#) to use this and other attributes discussed in this article. The attribute is applied to the property, not the `set` accessor. The `AllowNull` attribute specifies *pre-conditions*, and only applies to arguments. The `get` accessor has a return value, but no parameters. Therefore, the `AllowNull` attribute only applies to the `set` accessor.

The preceding example demonstrates what to look for when adding the `AllowNull` attribute on an argument:

1. The general contract for that variable is that it shouldn't be `null`, so you want a non-nullable reference type.
2. There are scenarios for the parameter to be `null`, though they aren't the most common usage.

Most often you'll need this attribute for properties, or `in`, `out`, and `ref` arguments. The `AllowNull` attribute is the best choice when a variable is typically non-null, but you need to allow `null` as a precondition.

Contrast that with scenarios for using `DisallowNull`: You use this attribute to specify that an argument of a nullable reference type shouldn't be `null`. Consider a property where `null` is the default value, but clients can only set it to a non-null value. Consider the following code:

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

The preceding code is the best way to express your design that the `ReviewComment` could be `null`, but can't be set to `null`. Once this code is nullable aware, you can express this concept more clearly to callers using the [System.Diagnostics.CodeAnalysis.DisallowNullAttribute](#):

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

In a nullable context, the `ReviewComment` `get` accessor could return the default value of `null`. The compiler warns that it must be checked before access. Furthermore, it warns callers that, even though it could be `null`, callers shouldn't explicitly set it to `null`. The `DisallowNull` attribute also specifies a *pre-condition*, it doesn't affect the `get` accessor. You use the `DisallowNull` attribute when you observe these characteristics about:

1. The variable could be `null` in core scenarios, often when first instantiated.
2. The variable shouldn't be explicitly set to `null`.

These situations are common in code that was originally *null oblivious*. It may be that object properties are set in two distinct initialization operations. It may be that some properties are set only after some asynchronous work has completed.

The `AllowNull` and `DisallowNull` attributes enable you to specify that preconditions on variables may not match the nullable annotations on those variables. These provide more detail about the characteristics of your API. This additional information helps callers use your API correctly. Remember you specify preconditions using the following attributes:

- [AllowNull](#): A non-nullable argument may be null.
- [DisallowNull](#): A nullable argument should never be null.

Specify post-conditions: `MaybeNull` and `NotNull`

Suppose you have a method with the following signature:

```
public Customer FindCustomer(string lastName, string firstName)
```

You've likely written a method like this to return `null` when the name sought wasn't found. The `null` clearly indicates that the record wasn't found. In this example, you'd likely change the return type from `Customer` to `Customer?`. Declaring the return value as a nullable reference type specifies the intent of this API clearly.

For reasons covered under [Generic definitions and nullability](#) that technique doesn't work with generic methods. You may have a generic method that follows a similar pattern:

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

You can't specify that the return value is `T?`. The method returns `null` when the sought item isn't found. Since you can't declare a `T?` return type, you add the `MaybeNull` annotation to the method return:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The preceding code informs callers that the contract implies a non-nullable type, but the return value *may* actually be null. Use the `MaybeNull` attribute when your API should be a non-nullable type, typically a generic type parameter, but there may be instances where `null` would be returned.

You can also specify that a return value or an `out` or `ref` argument isn't null even though the type is a nullable reference type. Consider a method that ensures an array is large enough to hold a number of elements. If the argument doesn't have capacity, the routine would allocate a new array and copy all the existing elements into it. If the argument is `null`, the routine would allocate new storage. If there's sufficient capacity, the routine does nothing:

```
public void EnsureCapacity<T>(ref T[] storage, int size)
```

You could call this routine as follows:

```
// messages has the default value (null) when EnsureCapacity is called:
EnsureCapacity<string>(ref messages, 10);
// messages is not null.
EnsureCapacity<string>(messages, 50);
```

After enabling null reference types, you want to ensure that the preceding code compiles without warnings. When the method returns, the `storage` argument is guaranteed to be not null. However, it's acceptable to call `EnsureCapacity` with a null reference. You can make `storage` a nullable reference type, and add the `NotNull` post-condition to the parameter declaration:

```
public void EnsureCapacity<T>([NotNull] ref T[]? storage, int size)
```

The preceding code expresses the existing contract clearly: Callers can pass a variable with the `null` value, but the return value is guaranteed to never be null. The `NotNull` attribute is most useful for `ref` and `out` arguments where `null` may be passed as an argument, but that argument is guaranteed to be not null when the method returns.

You specify unconditional postconditions using the following attributes:

- `MaybeNull`: A non-nullable return value may be null.
- `NotNull`: A nullable return value will never be null.

Specify conditional post-conditions: `NotNullWhen`, `MaybeNullWhen`, and `NotNullIfNotNull`

You're likely familiar with the `string` method `String.IsNullOrEmpty(String)`. This method returns `true` when the argument is null or an empty string. It's a form of null-check: Callers don't need to null-check the argument if the method returns `false`. To make a method like this nullable aware, you'd set the argument to a nullable reference type, and add the `NotNullWhen` attribute:

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value);
```

That informs the compiler that any code where the return value is `false` doesn't need null checks. The addition of the attribute informs the compiler's static analysis that `IsNullOrEmpty` performs the necessary null check: when it returns `false`, the argument isn't `null`.

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

The `String.IsNullOrEmpty(String)` method will be annotated as shown above for .NET Core 3.0. You may have similar methods in your codebase that check the state of objects for null values. The compiler won't recognize custom null check methods, and you'll need to add the annotations yourself. When you add the attribute, the compiler's static analysis knows when the tested variable has been null checked.

Another use for these attributes is the `Try*` pattern. The postconditions for `ref` and `out` variables are communicated through the return value. Consider this method shown earlier:

```
bool TryGetMessage(string key, out string message)
```

The preceding method follows a typical .NET idiom: the return value indicates if `message` was set to the found value or, if no message is found, to the default value. If the method returns `true`, the value of `message` isn't null; otherwise, the method sets `message` to null.

You can communicate that idiom using the `NotNullWhen` attribute. When you update the signature for nullable reference types, make `message` a `string?` and add an attribute:

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
```

In the preceding example, the value of `message` is known to be not null when `TryGetMessage` returns `true`. You should annotate similar methods in your codebase in the same way: the arguments could be `null`, and are known to be not null when the method returns `true`.

There's one final attribute you may also need. Sometimes the null state of a return value depends on the null state of one or more arguments. These methods will return a non-null value whenever certain arguments aren't `null`. To correctly annotate these methods, you use the `NotNullIfNotNull` attribute. Consider the following method:

```
string GetTopLevelDomainFromFullUrl(string url);
```

If the `url` argument isn't null, the output isn't `null`. Once nullable references are enabled, that signature works correctly, provided your API never accepts a null argument. However, if the argument could be null, then return value could also be null. You could change the signature to the following code:

```
string? GetTopLevelDomainFromFullUrl(string? url);
```

That also works, but will often force callers to implement extra `null` checks. The contract is that the return value would be `null` only when the argument `url` is `null`. To express that contract, you would annotate this

method as shown in the following code:

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url);
```

The return value and the argument have both been annotated with the `?` indicating that either could be `null`. The attribute further clarifies that the return value won't be null when the `url` argument isn't `null`.

You specify conditional postconditions using these attributes:

- [MaybeNullWhen](#): A non-nullable argument may be null when the method returns the specified `bool` value.
- [NotNullWhen](#): A nullable argument won't be null when the method returns the specified `bool` value.
- [NotNullIfNotNull](#): A return value isn't null if the argument for the specified parameter isn't null.

Constructor helper methods: [MemberNotNull](#) and [MemberNotNullWhen](#)

These attributes specify your intent when you've refactored common code from constructors into helper methods. The C# compiler analyzes constructors and field initializers to make sure that all non-nullable reference fields have been initialized before each constructor returns. However, the C# compiler doesn't track field assignments through all helper methods. The compiler issues warning `CS8618` when fields aren't initialized directly in the constructor, but rather in a helper method. You add the [MemberNotNullAttribute](#) to a method declaration and specify the fields that are initialized to a non-null value in the method. For example, consider the following example:

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

You can specify multiple field names as arguments to the [MemberNotNull](#) attribute constructor.

The [MemberNotNullWhenAttribute](#) has a `bool` argument. You use [MemberNotNullWhen](#) in situations where your helper method returns a `bool` indicating whether your helper method initialized fields.

Verify unreachable code

Some methods, typically exception helpers or other utility methods, always exit by throwing an exception. Or, a helper may throw an exception based on the value of a Boolean argument.

In the first case, you can add the [DoesNotReturn](#) attribute to the method declaration. The compiler helps you in

three ways. First, the compiler issues a warning if there's a path where the method can exit without throwing an exception. Second, the compiler marks any code after a call to that method as *unreachable*, until an appropriate `catch` clause is found. Third, the unreachable code won't affect any null states. Consider this method:

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (!isInitialized)
    {
        FailFast();
    }

    // unreachable code:
    _field = containedField;
}
```

In the second case, you add the `DoesNotReturnIf` attribute to a Boolean parameter of the method. You can modify the previous example as follows:

```
private void FailFast([DoesNotReturnIf(false)] bool isValid)
{
    if (!isValid)
    {
        throw new InvalidOperationException();
    }
}

public void SetState(object containedField)
{
    FailFast(isInitialized);

    // unreachable code when "isInitialized" is false:
    _field = containedField;
}
```

Summary

IMPORTANT

The official documentation tracks the latest C# version. We are currently writing for C# 9.0. Depending on the version of C# you're using, various features may not be available. The *default* C# version for your project is based on the target framework. For more information, see [C# language versioning defaults](#).

Adding nullable reference types provides an initial vocabulary to describe your APIs expectations for variables that could be `null`. The attributes provide a richer vocabulary to describe the null state of variables as preconditions and postconditions. These attributes more clearly describe your expectations and provide a better experience for the developers using your APIs.

As you update libraries for a nullable context, add these attributes to guide users of your APIs to the correct usage. These attributes help you fully describe the null-state of arguments and return values:

- [AllowNull](#): A non-nullable argument may be null.
- [DisallowNull](#): A nullable argument should never be null.

- [MaybeNull](#): A non-nullable return value may be null.
- [NotNull](#): A nullable return value will never be null.
- [MaybeNullWhen](#): A non-nullable argument may be null when the method returns the specified `bool` value.
- [NotNullWhen](#): A nullable argument won't be null when the method returns the specified `bool` value.
- [NotNullIfNotNull](#): A return value isn't null if the argument for the specified parameter isn't null.
- [DoesNotReturn](#): A method never returns. In other words, it always throws an exception.
- [DoesNotReturnIf](#): This method never returns if the associated `bool` parameter has the specified value.

Reserved attributes: Miscellaneous

3/25/2021 • 7 minutes to read • [Edit Online](#)

These attributes can be applied to elements in your code. They add semantic meaning to those elements. The compiler uses those semantic meanings to alter its output and report possible mistakes by developers using your code.

Conditional attribute

The `conditional` attribute makes the execution of a method dependent on a preprocessing identifier. The `Conditional` attribute is an alias for [ConditionalAttribute](#), and can be applied to a method or an attribute class.

In the following example, `Conditional` is applied to a method to enable or disable the display of program-specific diagnostic information:

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

If the `TRACE_ON` identifier isn't defined, the trace output isn't displayed. Explore for yourself in the interactive window.

The `conditional` attribute is often used with the `DEBUG` identifier to enable trace and logging features for debug builds but not in release builds, as shown in the following example:

```
[Conditional("DEBUG")]
static void DebugMethod()
{}
```

When a method marked conditional is called, the presence or absence of the specified preprocessing symbol determines whether the call is included or omitted. If the symbol is defined, the call is included; otherwise, the call is omitted. A conditional method must be a method in a class or struct declaration and must have a `void`

return type. Using `Conditional` is cleaner, more elegant, and less error-prone than enclosing methods inside `#if...#endif` blocks.

If a method has multiple `Conditional` attributes, a call to the method is included if at one or more conditional symbols is defined (the symbols are logically linked together by using the OR operator). In the following example, the presence of either `A` or `B` results in a method call:

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Using `Conditional` with attribute classes

The `Conditional` attribute can also be applied to an attribute class definition. In the following example, the custom attribute `Documentation` will only add information to the metadata if `DEBUG` is defined.

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Obsolete attribute

The `Obsolete` attribute marks a code element as no longer recommended for use. Use of an entity marked `obsolete` generates a warning or an error. The `Obsolete` attribute is a single-use attribute and can be applied to any entity that allows attributes. `Obsolete` is an alias for `ObsoleteAttribute`.

In the following example the `Obsolete` attribute is applied to class `A` and to method `B.oldMethod`. Because the second argument of the attribute constructor applied to `B.oldMethod` is set to `true`, this method will cause a compiler error, whereas using class `A` will just produce a warning. Calling `B.NewMethod`, however, produces no warning or error. For example, when you use it with the previous definitions, the following code generates two warnings and one error:

```

using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }

    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}

```

The string provided as the first argument to the attribute constructor will be displayed as part of the warning or error. Two warnings for class `A` are generated: one for the declaration of the class reference, and one for the class constructor. The `Obsolete` attribute can be used without arguments, but including an explanation what to use instead is recommended.

`AttributeUsage` attribute

The `AttributeUsage` attribute determines how a custom attribute class can be used. [AttributeUsageAttribute](#) is an attribute you apply to custom attribute definitions. The `AttributeUsage` attribute enables you to control:

- Which program elements attribute may be applied to. Unless you restrict its usage, an attribute may be applied to any of the following program elements:
 - assembly
 - module
 - field
 - event
 - method
 - param
 - property
 - return
 - type
- Whether an attribute can be applied to a single program element multiple times.
- Whether attributes are inherited by derived classes.

The default settings look like the following example when applied explicitly:

```
[AttributeUsage(AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)]
class NewAttribute : Attribute { }
```

In this example, the `NewAttribute` class can be applied to any supported program element. But it can be applied only once to each entity. The attribute is inherited by derived classes when applied to a base class.

The `AllowMultiple` and `Inherited` arguments are optional, so the following code has the same effect:

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

The first `AttributeUsageAttribute` argument must be one or more elements of the `AttributeTargets` enumeration. Multiple target types can be linked together with the OR operator, like the following example shows:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Beginning in C# 7.3, attributes can be applied to either the property or the backing field for an auto-implemented property. The attribute applies to the property, unless you specify the `field` specifier on the attribute. Both are shown in the following example:

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; }

    // Attribute attached to backing field:
    [field:NewPropertyOrField]
    public string Description { get; set; }
}
```

If the `AllowMultiple` argument is `true`, then the resulting attribute can be applied more than once to a single entity, as shown in the following example:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

In this case, `MultiUseAttribute` can be applied repeatedly because `AllowMultiple` is set to `true`. Both formats shown for applying multiple attributes are valid.

If `Inherited` is `false`, then the attribute isn't inherited by classes derived from an attributed class. For example:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

In this case `NonInheritedAttribute` isn't applied to `DClass` via inheritance.

ModuleInitializer attribute

Starting with C# 9, the `ModuleInitializer` attribute marks a method that the runtime calls when the assembly loads. `ModuleInitializer` is an alias for [ModuleInitializerAttribute](#).

The `ModuleInitializer` attribute can only be applied to a method that:

- Is static.
- Is parameterless.
- Returns `void`.
- Is accessible from the containing module, that is, `internal` or `public`.
- Isn't a generic method.
- Isn't contained in a generic class.
- Isn't a local function.

The `ModuleInitializer` attribute can be applied to multiple methods. In that case, the order in which the runtime calls them is deterministic but not specified.

The following example illustrates use of multiple module initializer methods. The `Init1` and `Init2` methods run before `Main`, and each adds a string to the `Text` property. So when `Main` runs, the `Text` property already has strings from both initializer methods.

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Source code generators sometimes need to generate initialization code. Module initializers provide a standard place for that code to reside.

SkipLocalsInit attribute

Starting in C# 9, the `SkipLocalsInit` attribute prevents the compiler from setting the `.locals init` flag when emitting to metadata. The `SkipLocalsInit` attribute is a single-use attribute and can be applied to a method, a property, a class, a struct, an interface, or a module, but not to an assembly. `SkipLocalsInit` is an alias for [SkipLocalsInitAttribute](#).

The `.locals init` flag causes the CLR to initialize all of the local variables declared in a method to their default values. Since the compiler also makes sure that you never use a variable before assigning some value to it, `.locals init` is typically not necessary. However, the extra zero-initialization may have measurable performance impact in some scenarios, such as when you use `stackalloc` to allocate an array on the stack. In those cases, you can add the `SkipLocalsInit` attribute. If applied to a method directly, the attribute affects that method and all its nested functions, including lambdas and local functions. If applied to a type or module, it affects all methods nested inside. This attribute does not affect abstract methods, but it does affect code generated for the implementation.

This attribute requires the [AllowUnsafeBlocks](#) compiler option. This is to signal that in some cases code could view unassigned memory (for example, reading from uninitialized stack-allocated memory).

The following example illustrates the effect of `SkipLocalsInit` attribute on a method that uses `stackalloc`. The method displays whatever was in memory when the array of integers was allocated.

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

To try this code yourself, set the `AllowUnsafeBlocks` compiler option in your `.csproj` file:

```
<PropertyGroup>
    ...
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

See also

- [Attribute](#)
- [System.Reflection](#)
- [Attributes](#)
- [Reflection](#)

Unsafe code and pointer types

4/6/2021 • 13 minutes to read • [Edit Online](#)

Most of the C# code you write is "verifiably safe code." *Verifiably safe code* means .NET tools can verify that the code is safe. In general, safe code doesn't directly access memory using pointers. It also doesn't allocate raw memory. It creates managed objects instead.

C# supports an `unsafe` context, in which you may write *unverifiable* code. In an `unsafe` context, code may use pointers, allocate and free blocks of memory, and call methods using function pointers. Unsafe code in C# isn't necessarily dangerous; it's just code whose safety cannot be verified.

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code may increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- The code that contains unsafe blocks must be compiled with the [AllowUnsafeBlocks](#) compiler option.

Pointer types

In an unsafe context, a type may be a pointer type, in addition to a value type, or a reference type. A pointer type declaration takes one of the following forms:

```
type* identifier;  
void* identifier; //allowed but not recommended
```

The type specified before the `*` in a pointer type is called the **referent type**. Only an [unmanaged type](#) can be a referent type.

Pointer types don't inherit from `object` and no conversions exist between pointer types and `object`. Also, boxing and unboxing don't support pointers. However, you can convert between different pointer types and between pointer types and integral types.

When you declare multiple pointers in the same declaration, you write the asterisk (`*`) together with the underlying type only. It isn't used as a prefix to each pointer name. For example:

```
int* p1, p2, p3; // Ok  
int *p1, *p2, *p3; // Invalid in C#
```

A pointer can't point to a reference or to a `struct` that contains references, because an object reference can be garbage collected even if a pointer is pointing to it. The garbage collector doesn't keep track of whether an object is being pointed to by any pointer types.

The value of the pointer variable of type `MyType*` is the address of a variable of type `MyType`. The following are examples of pointer type declarations:

- `int* p : p` is a pointer to an integer.
- `int** p : p` is a pointer to a pointer to an integer.
- `int*[] p : p` is a single-dimensional array of pointers to integers.

- `char* p` : `p` is a pointer to a char.
- `void* p` : `p` is a pointer to an unknown type.

The pointer indirection operator `*` can be used to access the contents at the location pointed to by the pointer variable. For example, consider the following declaration:

```
int* myVariable;
```

The expression `*myVariable` denotes the `int` variable found at the address contained in `myVariable`.

There are several examples of pointers in the articles on the [fixed statement](#). The following example uses the `unsafe` keyword and the `fixed` statement, and shows how to increment an interior pointer. You can paste this code into the Main function of a console application to run it. These examples must be compiled with the [AllowUnsafeBlocks](#) compiler option set.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("-----");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("-----");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
-----
10
11
12
-----
12
*/
```

You can't apply the indirection operator to a pointer of type `void*`. However, you can use a cast to convert a void pointer to any other pointer type, and vice versa.

A pointer can be `null`. Applying the indirection operator to a null pointer causes an implementation-defined behavior.

Passing pointers between methods can cause undefined behavior. Consider a method that returns a pointer to a local variable through an `in`, `out`, or `ref` parameter or as the function result. If the pointer was set in a fixed block, the variable to which it points may no longer be fixed.

The following table lists the operators and statements that can operate on pointers in an unsafe context:

OPERATOR/STATEMENT	USE
<code>*</code>	Performs pointer indirection.
<code>-></code>	Accesses a member of a struct through a pointer.
<code>[]</code>	Indexes a pointer.
<code>&</code>	Obtains the address of a variable.
<code>++</code> and <code>--</code>	Increments and decrements pointers.
<code>+</code> and <code>-</code>	Performs pointer arithmetic.
<code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , and <code>>=</code>	Compares pointers.
<code>stackalloc</code>	Allocates memory on the stack.
<code>fixed</code> statement	Temporarily fixes a variable so that its address may be found.

For more information about pointer-related operators, see [Pointer-related operators](#).

Any pointer type can be implicitly converted to a `void*` type. Any pointer type can be assigned the value `null`. Any pointer type can be explicitly converted to any other pointer type using a cast expression. You can also convert any integral type to a pointer type, or any pointer type to an integral type. These conversions require an explicit cast.

The following example converts an `int*` to a `byte*`. Notice that the pointer points to the lowest addressed byte of the variable. When you successively increment the result, up to the size of `int` (4 bytes), you can display the remaining bytes of the variable.

```

int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
       The 4 bytes of the integer: 00 04 00 00
       The value of the integer: 1024
    */
}

```

Fixed-size buffers

In C#, you can use the `fixed` statement to create a buffer with a fixed size array in a data structure. Fixed size buffers are useful when you write methods that interoperate with data sources from other languages or platforms. The fixed array can take any attributes or modifiers that are allowed for regular struct members. The only restriction is that the array type must be `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`.

```
private fixed char name[30];
```

In safe code, a C# struct that contains an array doesn't contain the array elements. The struct contains a reference to the elements instead. You can embed an array of fixed size in a `struct` when it's used in an `unsafe` code block.

The size of the following `struct` doesn't depend on the number of elements in the array, since `pathName` is a reference:

```

public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

A `struct` can contain an embedded array in unsafe code. In the following example, the `fixedBuffer` array has a fixed size. You use a `fixed` statement to establish a pointer to the first element. You access the elements of the array through this pointer. The `fixed` statement pins the `fixedBuffer` instance field to a specific location in memory.

```

internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}

```

The size of the 128 element `char` array is 256 bytes. Fixed size `char` buffers always take 2 bytes per character, regardless of the encoding. This array size is the same even when `char` buffers are marshaled to API methods or structs with `CharSet = CharSet.Auto` or `CharSet = CharSet.Ansi`. For more information, see [CharSet](#).

The preceding example demonstrates accessing `fixed` fields without pinning, which is available starting with C# 7.3.

Another common fixed-size array is the `bool` array. The elements in a `bool` array are always 1 byte in size. `bool` arrays aren't appropriate for creating bit arrays or buffers.

Fixed size buffers are compiled with the [System.Runtime.CompilerServices.UnsafeValueTypeAttribute](#), which instructs the common language runtime (CLR) that a type contains an unmanaged array that can potentially overflow. Memory allocated using `stackalloc` also automatically enables buffer overrun detection features in the CLR. The previous example shows how a fixed size buffer could exist in an `unsafe struct`.

```

internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

```

The compiler-generated C# for `Buffer` is attributed as follows:

```

internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}

```

Fixed size buffers differ from regular arrays in the following ways:

- May only be used in an `unsafe` context.
- May only be instance fields of structs.
- They're always vectors, or one-dimensional arrays.
- The declaration should include the length, such as `fixed char id[8]`. You can't use `fixed char id[]`.

How to use pointers to copy an array of bytes

The following example uses pointers to copy bytes from one array to another.

This example uses the `unsafe` keyword, which enables you to use pointers in the `Copy` method. The `fixed` statement is used to declare pointers to the source and destination arrays. The `fixed` statement *pins* the location of the source and destination arrays in memory so that they will not be moved by garbage collection. The memory blocks for the arrays are unpinned when the `fixed` block is completed. Because the `Copy` method in this example uses the `unsafe` keyword, it must be compiled with the [AllowUnsafeBlocks](#) compiler option.

This example accesses the elements of both arrays using indices rather than a second unmanaged pointer. The declaration of the `pSource` and `pTarget` pointers pins the arrays. This feature is available starting with C# 7.3.

```

static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.

```

```

// Collection.

fixed (byte* pSource = source, pTarget = target)
{
    // Copy the specified number of bytes from source to target.
    for (int i = 0; i < count; i++)
    {
        pTarget[targetOffset + i] = pSource[sourceOffset + i];
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
       The first 10 elements of the original are:
       0 1 2 3 4 5 6 7 8 9

       The first 10 elements of the copy are:
       0 1 2 3 4 5 6 7 8 9

       The first 10 elements of the copy are:
       90 91 92 93 94 95 96 97 98 99
    */
}

```

Function pointers

C# provides `delegate` types to define safe function pointer objects. Invoking a delegate involves instantiating a type derived from `System.Delegate` and making a virtual method call to its `Invoke` method. This virtual call uses the `callvirt` IL instruction. In performance critical code paths, using the `calli` IL instruction is more efficient.

You can define a function pointer using the `delegate*` syntax. The compiler will call the function using the `calli` instruction rather than instantiating a `delegate` object and calling `Invoke`. The following code declares two methods that use a `delegate` or a `delegate*` to combine two objects of the same type. The first method uses a `System.Func<T1,T2,TResult>` delegate type. The second method uses a `delegate*` declaration with the same parameters and return type:

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
```

The following code shows how you would declare a static local function and invoke the `UnsafeCombine` method using a pointer to that local function:

```
static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);
```

The preceding code illustrates several of the rules on the function accessed as a function pointer:

- Function pointers can only be declared in an `unsafe` context.
- Methods that take a `delegate*` (or return a `delegate*`) can only be called in an `unsafe` context.
- The `&` operator to obtain the address of a function is allowed only on `static` functions. (This rule applies to both member functions and local functions).

The syntax has parallels with declaring `delegate` types and using pointers. The `*` suffix on `delegate` indicates the declaration is a *function pointer*. The `&` when assigning a method group to a function pointer indicates the operation takes the address of the method.

You can specify the calling convention for a `delegate*` using the keywords `managed` and `unmanaged`. In addition, for `unmanaged` function pointers, you can specify the calling convention. The following declarations show examples of each. The first declaration uses the `managed` calling convention, which is the default. The next three use an `unmanaged` calling convention. Each specifies one of the ECMA 335 calling conventions: `Cdecl`, `Stdcall`, `Fastcall`, or `Thiscall`. The last declarations uses the `unmanaged` calling convention, instructing the CLR to pick the default calling convention for the platform. The CLR will choose the calling convention at run time.

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
```

You can learn more about function pointers in the [Function pointer proposal for C# 9.0](#).

C# language specification

For more information, see the [Unsafe code](#) chapter of the [C# language specification](#).

C# preprocessor directives

3/27/2021 • 12 minutes to read • [Edit Online](#)

Although the compiler doesn't have a separate preprocessor, the directives described in this section are processed as if there were one. You use them to help in conditional compilation. Unlike C and C++ directives, you can't use these directives to create macros. A preprocessor directive must be the only instruction on a line.

Nullable context

The `#nullable` preprocessor directive sets the *nullable annotation context* and *nullable warning context*. This directive controls whether nullable annotations have effect, and whether nullability warnings are given. Each context is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code). The `#nullable` directive controls the annotation and warning contexts and takes precedence over the project-level settings. A directive sets the context(s) it controls until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable` : Sets the nullable annotation and warning contexts to *disabled*.
- `#nullable enable` : Sets the nullable annotation and warning contexts to *enabled*.
- `#nullable restore` : Restores the nullable annotation and warning contexts to project settings.
- `#nullable disable annotations` : Sets the nullable annotation context to *disabled*.
- `#nullable enable annotations` : Sets the nullable annotation context to *enabled*.
- `#nullable restore annotations` : Restores the nullable annotation context to project settings.
- `#nullable disable warnings` : Sets the nullable warning context to *disabled*.
- `#nullable enable warnings` : Sets the nullable warning context to *enabled*.
- `#nullable restore warnings` : Restores the nullable warning context to project settings.

Conditional compilation

You use four preprocessor directives to control conditional compilation:

- `#if` : Opens a conditional compilation, where code is compiled only if the specified symbol is defined.
- `#elif` : Closes the preceding conditional compilation and opens a new conditional compilation based on if the specified symbol is defined.
- `#else` : Closes the preceding conditional compilation and opens a new conditional compilation if the previous specified symbol isn't defined.
- `#endif` : Closes the preceding conditional compilation.

When the C# compiler finds an `#if` directive, followed eventually by an `#endif` directive, it compiles the code between the directives only if the specified symbol is defined. Unlike C and C++, you can't assign a numeric value to a symbol. The `#if` statement in C# is Boolean and only tests whether the symbol has been defined or not. For example:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

You can use the operators `==` (equality) and `!=` (inequality) to test for the `bool` values `true` or `false`. `true` means the symbol is defined. The statement `#if DEBUG` has the same meaning as `#if (DEBUG == true)`. You can use the `&&` (and), `||` (or), and `!` (not) operators to evaluate whether multiple symbols have been defined. You can also group symbols and operators with parentheses.

`#if`, along with the `#else`, `#elif`, `#endif`, `#define`, and `#undef` directives, lets you include or exclude code based on the existence of one or more symbols. Conditional compilation can be useful when compiling code for a debug build or when compiling for a specific configuration.

A conditional directive beginning with an `#if` directive must explicitly be terminated with an `#endif` directive. `#define` lets you define a symbol. By using the symbol as the expression passed to the `#if` directive, the expression evaluates to `true`. You can also define a symbol with the **DefineConstants** compiler option. You can undefine a symbol with `#undef`. The scope of a symbol created with `#define` is the file in which it was defined. A symbol that you define with **DefineConstants** or with `#define` doesn't conflict with a variable of the same name. That is, a variable name shouldn't be passed to a preprocessor directive, and a symbol can only be evaluated by a preprocessor directive.

`#elif` lets you create a compound conditional directive. The `#elif` expression will be evaluated if neither the preceding `#if` nor any preceding, optional, `#elif` directive expressions evaluate to `true`. If an `#elif` expression evaluates to `true`, the compiler evaluates all the code between the `#elif` and the next conditional directive. For example:

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` lets you create a compound conditional directive, so that, if none of the expressions in the preceding `#if` or (optional) `#elif` directives evaluate to `true`, the compiler will evaluate all code between `#else` and the next `#endif`. `#endif` (`#endif`) must be the next preprocessor directive after `#else`.

`#endif` specifies the end of a conditional directive, which began with the `#if` directive.

The build system is also aware of predefined preprocessor symbols representing different **target frameworks** in SDK-style projects. They're useful when creating applications that can target more than one .NET version.

TARGET FRAMEWORKS	SYMBOLS
.NET Framework	<code>NETFRAMEWORK</code> , <code>NET48</code> , <code>NET472</code> , <code>NET471</code> , <code>NET47</code> , <code>NET462</code> , <code>NET461</code> , <code>NET46</code> , <code>NET452</code> , <code>NET451</code> , <code>NET45</code> , <code>NET40</code> , <code>NET35</code> , <code>NET20</code>
.NET Standard	<code>NETSTANDARD</code> , <code>NETSTANDARD2_1</code> , <code>NETSTANDARD2_0</code> , <code>NETSTANDARD1_6</code> , <code>NETSTANDARD1_5</code> , <code>NETSTANDARD1_4</code> , <code>NETSTANDARD1_3</code> , <code>NETSTANDARD1_2</code> , <code>NETSTANDARD1_1</code> , <code>NETSTANDARD1_0</code>
.NET 5 (and .NET Core)	<code>NET</code> , <code>NET5_0</code> , <code>NETCOREAPP</code> , <code>NETCOREAPP3_1</code> , <code>NETCOREAPP3_0</code> , <code>NETCOREAPP2_2</code> , <code>NETCOREAPP2_1</code> , <code>NETCOREAPP2_0</code> , <code>NETCOREAPP1_1</code> , <code>NETCOREAPP1_0</code>

NOTE

For traditional, non-SDK-style projects, you have to manually configure the conditional compilation symbols for the different target frameworks in Visual Studio via the project's properties pages.

Other predefined symbols include the `DEBUG` and `TRACE` constants. You can override the values set for the project using `#define`. The `DEBUG` symbol, for example, is automatically set depending on your build configuration properties ("Debug" or "Release" mode).

The following example shows you how to define a `MYTEST` symbol on a file and then test the values of the `MYTEST` and `DEBUG` symbols. The output of this example depends on whether you built the project on **Debug** or **Release** configuration mode.

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

The following example shows you how to test for different target frameworks so you can use newer APIs when possible:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Defining symbols

You use the following two preprocessor directives to define or undefine symbols for conditional compilation:

- `#define` : Define a symbol.
- `#undef` : undefine a symbol.

You use `#define` to define a symbol. When you use the symbol as the expression that's passed to the `#if` directive, the expression will evaluate to `true`, as the following example shows:

```
#define VERBOSE

#if VERBOSE
    Console.WriteLine("Verbose output version");
#endif
```

NOTE

The `#define` directive cannot be used to declare constant values as is typically done in C and C++. Constants in C# are best defined as static members of a class or struct. If you have several such constants, consider creating a separate "Constants" class to hold them.

Symbols can be used to specify conditions for compilation. You can test for the symbol with either `#if` or `#elif`. You can also use the [ConditionalAttribute](#) to perform conditional compilation. You can define a symbol, but you can't assign a value to a symbol. The `#define` directive must appear in the file before you use any instructions that aren't also preprocessor directives. You can also define a symbol with the [DefineConstants](#) compiler option. You can undefine a symbol with `#undef`.

Defining regions

You can define regions of code that can be collapsed in an outline using the following two preprocessor directives:

- `#region` : Start a region.
- `#endregion` : End a region

`#region` lets you specify a block of code that you can expand or collapse when using the [outlining](#) feature of the code editor. In longer code files, it's convenient to collapse or hide one or more regions so that you can focus on the part of the file that you're currently working on. The following example shows how to define a region:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

A `#region` block must be terminated with an `#endregion` directive. A `#region` block can't overlap with an `#if` block. However, a `#region` block can be nested in an `#if` block, and an `#if` block can be nested in a `#region` block.

Error and warning information

You instruct the compiler to generate user-defined compiler errors and warnings, and control line information using the following directives:

- `#error` : Generate a compiler error with a specified message.
- `#warning` : Generate a compiler warning, with a specific message.
- `#line` : Change the line number printed with compiler messages.

`#error` lets you generate a [CS1029](#) user-defined error from a specific location in your code. For example:

```
#error Deprecated code in this method.
```

NOTE

The compiler treats `#error version` in a special way and reports a compiler error, CS8304, with a message containing the used compiler and language versions.

`#warning` lets you generate a [CS1030](#) level one compiler warning from a specific location in your code. For example:

```
#warning Deprecated code in this method.
```

`#line` lets you modify the compiler's line numbering and (optionally) the file name output for errors and warnings.

The following example shows how to report two warnings associated with line numbers. The `#line 200` directive forces the next line's number to be 200 (although the default is #6), and until the next `#line` directive, the filename will be reported as "Special". The `#line default` directive returns the line numbering to its default numbering, which counts the lines that were renumbered by the previous directive.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

Compilation produces the following output:

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

The `#line` directive might be used in an automated, intermediate step in the build process. For example, if lines were removed from the original source code file, but you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with `#line`.

The `#line hidden` directive hides the successive lines from the debugger, such that when the developer steps through the code, any lines between a `#line hidden` and the next `#line` directive (assuming that it isn't another `#line hidden` directive) will be stepped over. This option can also be used to allow ASP.NET to differentiate between user-defined and machine-generated code. Although ASP.NET is the primary consumer of this feature,

it's likely that more source generators will make use of it.

A `#line hidden` directive doesn't affect file names or line numbers in error reporting. That is, if the compiler finds an error in a hidden block, the compiler will report the current file name and line number of the error.

The `#line filename` directive specifies the file name you want to appear in the compiler output. By default, the actual name of the source code file is used. The file name must be in double quotation marks ("") and must be preceded by a line number.

The following example shows how the debugger ignores the hidden lines in the code. When you run the example, it will display three lines of text. However, when you set a break point, as shown in the example, and hit F10 to step through the code, the debugger ignores the hidden line. Even if you set a break point at the hidden line, the debugger will still ignore it.

```
// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point here.
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}
```

Pragmas

`#pragma` gives the compiler special instructions for the compilation of the file in which it appears. The instructions must be supported by the compiler. In other words, you can't use `#pragma` to create custom preprocessing instructions.

- `#pragma warning` : Enable or disable warnings.
- `#pragma checksum` : Generate a checksum.

```
#pragma pragma-name pragma-arguments
```

Where `pragma-name` is the name of a recognized pragma and `pragma-arguments` is the pragma-specific arguments.

`#pragma warning`

`#pragma warning` can enable or disable certain warnings.

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Where `warning-list` is a comma-separated list of warning numbers. The "CS" prefix is optional. When no warning numbers are specified, `disable` disables all warnings and `restore` enables all warnings.

NOTE

To find warning numbers in Visual Studio, build your project and then look for the warning numbers in the **Output** window.

The `disable` takes effect beginning on the next line of the source file. The warning is restored on the line following the `restore`. If there's no `restore` in the file, the warnings are restored to their default state at the first line of any later files in the same compilation.

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}

#pragma warning restore CS3021
[CLSCompliant(false)] // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

#pragma checksum

Generates checksums for source files to aid with debugging ASP.NET pages.

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Where `"filename"` is the name of the file that requires monitoring for changes or updates, `"{guid}"` is the Globally Unique Identifier (GUID) for the hash algorithm, and `"checksum_bytes"` is the string of hexadecimal digits representing the bytes of the checksum. Must be an even number of hexadecimal digits. An odd number of digits results in a compile-time warning, and the directive is ignored.

The Visual Studio debugger uses a checksum to make sure that it always finds the right source. The compiler computes the checksum for a source file, and then emits the output to the program database (PDB) file. The debugger then uses the PDB to compare against the checksum that it computes for the source file.

This solution doesn't work for ASP.NET projects, because the computed checksum is for the generated source file, rather than the .aspx file. To address this problem, `#pragma checksum` provides checksum support for ASP.NET pages.

When you create an ASP.NET project in Visual C#, the generated source file contains a checksum for the .aspx file, from which the source is generated. The compiler then writes this information into the PDB file.

If the compiler doesn't find a `#pragma checksum` directive in the file, it computes the checksum and writes the value to the PDB file.

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

C# Compiler Options

3/23/2021 • 2 minutes to read • [Edit Online](#)

This section describes the options interpreted by the C# compiler. There are two different ways to set compiler options in .NET projects:

- **Specify option in your `*.csproj` file:** You can add XML elements for any compiler option in your `*.csproj` file. The element name is the same as the compiler option. The value of the XML element sets the value of the compiler option. For more information on setting options in project files, see the article [MSBuild properties for .NET SDK Projects](#).
- **Using the Visual Studio Property pages:** Visual Studio provides property pages to edit build properties. To learn more about them, see [Manage project and solution properties - Windows](#) or [Manage project and solution properties - Mac](#).

.NET Framework projects

IMPORTANT

This section applies to .NET Framework projects only.

In addition to the mechanisms described above, you can set compiler options using two additional methods for .NET Framework projects:

- **Command line arguments for .NET Framework projects:** .NET Framework projects use `csc.exe` instead of `dotnet build` to build projects. You can specify command line arguments to `csc.exe` for .NET Framework projects.
- **Compiled ASP.NET pages:** .NET Framework projects use a section of the `web.config` file for compiling pages. For the new build system, and ASP.NET Core projects, options are taken from the project file.

The word for some compiler options changed from `csc.exe` and .NET Framework projects to the new MSBuild system. The new syntax is used throughout this section. Both versions are listed at the top of each page. For `csc.exe`, any arguments are listed following the option and a colon. For example, the `-doc` option would be:

```
-doc:DocFile.xml
```

You can invoke the C# compiler by typing the name of its executable file (`csc.exe`) at a command prompt.

For .NET Framework projects, you can also run `csc.exe` from the command line. Every compiler option is available in two forms: **-option** and **/option**. In .NET Framework web projects, you specify options for compiling code-behind in the `web.config` file. For more information, see [`<compiler>` Element](#).

If you use the **Developer Command Prompt for Visual Studio** window, all the necessary environment variables are set for you. For information on how to access this tool, see [Developer Command Prompt for Visual Studio](#).

The `csc.exe` executable file is usually located in the `Microsoft.NET\Framework\<Version>` folder under the `Windows` directory. Its location might vary depending on the exact configuration of a particular computer. If more than one version of .NET Framework is installed on your computer, you'll find multiple versions of this file. For more information about such installations, see [How to: determine which versions of the .NET Framework are installed](#).

C# Compiler Options for language feature rules

4/6/2021 • 6 minutes to read • [Edit Online](#)

The following options control how the compiler interprets language features. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **CheckForOverflowUnderflow** / `-checked` : Generate overflow checks.
- **AllowUnsafeBlocks** / `-unsafe` : Allow 'unsafe' code.
- **DefineConstants** / `-define` : Define conditional compilation symbol(s).
- **LangVersion** / `-langversion` : Specify language version such as `default` (latest major version), or `latest` (latest version, including minor versions).
- **Nullable** / `-nullable` : Enable nullable context, or nullable warnings.

CheckForOverflowUnderflow

The **CheckForOverflowUnderflow** option specifies whether an integer arithmetic statement that results in a value that is outside the range of the data type causes a run-time exception.

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

An integer arithmetic statement that is in the scope of a `checked` or `unchecked` keyword isn't subject to the effect of the **CheckForOverflowUnderflow** option. If an integer arithmetic statement that isn't in the scope of a `checked` or `unchecked` keyword results in a value outside the range of the data type, and **CheckForOverflowUnderflow** is `true`, that statement causes an exception at run time. If **CheckForOverflowUnderflow** is `false`, that statement doesn't cause an exception at run time. The default value for this option is `false`; overflow checking is disabled.

AllowUnsafeBlocks

The **AllowUnsafeBlocks** compiler option allows code that uses the `unsafe` keyword to compile. The default value for this option is `false`, meaning unsafe code is not allowed.

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

For more information about unsafe code, see [Unsafe Code and Pointers](#).

DefineConstants

The **DefineConstants** option defines symbols in all source code files of your program.

```
<DefineConstants>name;name2</DefineConstants>
```

This option specifies the names of one or more symbols that you want to define. The **DefineConstants** option has the same effect as the `#define` preprocessor directive except that the compiler option is in effect for all files in the project. A symbol remains defined in a source file until an `#undef` directive in the source file removes the definition. When you use the `-define` option, an `#undef` directive in one file has no effect on other source code files in the project. You can use symbols created by this option with `#if`, `#else`, `#elif`, and `#endif` to compile source

files conditionally. The C# compiler itself defines no symbols or macros that you can use in your source code; all symbol definitions must be user-defined.

NOTE

The C# `#define` directive does not allow a symbol to be given a value, as in languages such as C++. For example, `#define` cannot be used to create a macro or to define a constant. If you need to define a constant, use an `enum` variable. If you want to create a C++ style macro, consider alternatives such as generics. Since macros are notoriously error-prone, C# disallows their use but provides safer alternatives.

LangVersion

Causes the compiler to accept only syntax that is included in the chosen C# language specification.

```
<LangVersion>9.0</LangVersion>
```

The following values are valid:

VALUE	MEANING
<code>preview</code>	The compiler accepts all valid language syntax from the latest preview version.
<code>latest</code>	The compiler accepts syntax from the latest released version of the compiler (including minor version).
<code>latestMajor</code> (<code>default</code>)	The compiler accepts syntax from the latest released major version of the compiler.
<code>9.0</code>	The compiler accepts only syntax that is included in C# 9.0 or lower.
<code>8.0</code>	The compiler accepts only syntax that is included in C# 8.0 or lower.
<code>7.3</code>	The compiler accepts only syntax that is included in C# 7.3 or lower.
<code>7.2</code>	The compiler accepts only syntax that is included in C# 7.2 or lower.
<code>7.1</code>	The compiler accepts only syntax that is included in C# 7.1 or lower.
<code>7</code>	The compiler accepts only syntax that is included in C# 7.0 or lower.
<code>6</code>	The compiler accepts only syntax that is included in C# 6.0 or lower.
<code>5</code>	The compiler accepts only syntax that is included in C# 5.0 or lower.

VALUE	MEANING
4	The compiler accepts only syntax that is included in C# 4.0 or lower.
3	The compiler accepts only syntax that is included in C# 3.0 or lower.
ISO-2 (or 2)	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
ISO-1 (or 1)	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

The default language version depends on the target framework for your application and the version of the SDK or Visual Studio installed. Those rules are defined in [C# language versioning](#).

Metadata referenced by your C# application isn't subject to the **LangVersion** compiler option.

Because each version of the C# compiler contains extensions to the language specification, **LangVersion** doesn't give you the equivalent functionality of an earlier version of the compiler.

Additionally, while C# version updates generally coincide with major .NET Framework releases, the new syntax and features aren't necessarily tied to that specific framework version. While the new features definitely require a new compiler update that is also released alongside the C# revision, each specific feature has its own minimum .NET API or common language runtime requirements that may allow it to run on downlevel frameworks by including NuGet packages or other libraries.

Regardless of which **LangVersion** setting you use, use the current version of the common language runtime to create your .exe or .dll. One exception is friend assemblies and **ModuleAssemblyName**, which work under `-langversion:ISO-1`.

For other ways to specify the C# language version, see [C# language versioning](#).

For information about how to set this compiler option programmatically, see [LanguageVersion](#).

C# language specification

VERSION	LINK	DESCRIPTION
C# 7.0 and later		Not currently available
C# 6.0	Link	C# Language Specification Version 6 - Unofficial Draft: .NET Foundation
C# 5.0	Download PDF	Standard ECMA-334 5th Edition
C# 3.0	Download DOC	C# Language Specification Version 3.0: Microsoft Corporation
C# 2.0	Download PDF	Standard ECMA-334 4th Edition
C# 1.2	Download DOC	C# Language Specification Version 1.2: Microsoft Corporation
C# 1.0	Download DOC	C# Language Specification Version 1.0: Microsoft Corporation

Minimum SDK version needed to support all language features

The following table lists the minimum versions of the SDK with the C# compiler that supports the corresponding language version:

C# VERSION	MINIMUM SDK VERSION
C# 8.0	Microsoft Visual Studio/Build Tools 2019, version 16.3, or .NET Core 3.0 SDK
C# 7.3	Microsoft Visual Studio/Build Tools 2017, version 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, version 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, version 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 or bundled .NET Framework 4.5 compiler
C# 4	Microsoft Visual Studio/Build Tools 2010 or bundled .NET Framework 4.0 compiler
C# 3	Microsoft Visual Studio/Build Tools 2008 or bundled .NET Framework 3.5 compiler
C# 2	Microsoft Visual Studio/Build Tools 2005 or bundled .NET Framework 2.0 compiler
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 or bundled .NET Framework 1.0 compiler

Nullable

The **Nullable** option lets you specify the nullable context.

```
<Nullable>enable</Nullable>
```

The argument must be one of `enable`, `disable`, `warnings`, or `annotations`. The `enable` argument enables the nullable context. Specifying `disable` will disable the nullable context. When providing the `warnings` argument the nullable warning context is enabled. When specifying the `annotations` argument, the nullable annotation context is enabled.

Flow analysis is used to infer the nullability of variables within executable code. The inferred nullability of a variable is independent of the variable's declared nullability. Method calls are analyzed even when they're conditionally omitted. For instance, `Debug.Assert` in release mode.

Invocation of methods annotated with the following attributes will also affect flow analysis:

- Simple pre-conditions: `AllowNullAttribute` and `DisallowNullAttribute`
- Simple post-conditions: `MaybeNullAttribute` and `NotNullAttribute`
- Conditional post-conditions: `MaybeNullWhenAttribute` and `NotNullWhenAttribute`

- [DoesNotReturnIfAttribute](#) (for example, `DoesNotReturnIf(false)` for [Debug.Assert](#)) and [DoesNotReturnAttribute](#)
- [NotNullIfNotNullAttribute](#)
- Member post-conditions: [MemberNotNullAttribute\(String\)](#) and [MemberNotNullAttribute\(String\[\]\)](#)

IMPORTANT

The global nullable context does not apply for generated code files. Regardless of this setting, the nullable context is *disabled* for any source file marked as generated. There are four ways a file is marked as generated:

1. In the `.editorconfig`, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with `TemporaryGeneratedFile_`
4. End the file name with `.designer.cs`, `.generated.cs`, `.g.cs`, or `.g.i.cs`.

Generators can opt-in using the `#nullable` preprocessor directive.

C# Compiler Options that control compiler output

3/27/2021 • 8 minutes to read • [Edit Online](#)

The following options control compiler output generation. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **DocumentationFile** / `-doc` : Generate XML doc file from `///` comments.
- **OutputAssembly** / `-out` : Specify the output assembly file.
- **PlatformTarget** / `-platform` : Specify the target platform CPU.
- **ProduceReferenceAssembly** / `-refout` : Generate a reference assembly.
- **TargetType** `-target` : Specify the type of the output assembly.

DocumentationFile

The **DocumentationFile** option allows you to place documentation comments in an XML file. To learn more about documenting your code, see [Recommended Tags for Documentation Comments](#). The value specifies the path to the output XML file. The XML file contains the comments in the source code files of the compilation.

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

The source code file that contains Main or top-level statements is output first into the XML. You'll often want to use the generated .xml file with [IntelliSense](#). The .xml filename must be the same as the assembly name. The .xml file must be in the same directory as the assembly. When the assembly is referenced in a Visual Studio project, the .xml file is found as well. For more information about generating code comments, see [Supplying Code Comments](#). Unless you compile with `<TargetType:Module>`, `file` will contain `<assembly>` and `</assembly>` tags specifying the name of the file containing the assembly manifest for the output file. For examples, see [How to use the XML documentation features](#).

NOTE

The **DocumentationFile** option applies to all files in the project. To disable warnings related to documentation comments for a specific file or section of code, use `#pragma warning`.

OutputAssembly

The **OutputAssembly** option specifies the name of the output file. The output path specifies the folder where compiler output is placed.

```
<OutputAssembly>folder</OutputAssembly>
```

Specify the full name and extension of the file you want to create. If you don't specify the name of the output file, MSBuild uses the name of the project to specify the name of the output assembly. Old style projects use the following rules:

- An .exe will take its name from the source code file that contains the `Main` method or top-level statements.
- A .dll or .netmodule will take its name from the first source code file.

Any modules produced as part of a compilation become files associated with any assembly also produced in the

compilation. Use [ildasm.exe](#) to view the assembly manifest to see the associated files.

The **OutputAssembly** compiler option is required in order for an exe to be the target of a [friend assembly](#).

PlatformTarget

Specifies which version of the CLR can run the assembly.

```
<PlatformTarget>anycpu</PlatformTarget>
```

- **anycpu** (default) compiles your assembly to run on any platform. Your application runs as a 64-bit process whenever possible and falls back to 32-bit when only that mode is available.
- **anycpu32bitpreferred** compiles your assembly to run on any platform. Your application runs in 32-bit mode on systems that support both 64-bit and 32-bit applications. You can specify this option only for projects that target .NET Framework 4.5 or later.
- **ARM** compiles your assembly to run on a computer that has an Advanced RISC Machine (ARM) processor.
- **ARM64** compiles your assembly to run by the 64-bit CLR on a computer that has an Advanced RISC Machine (ARM) processor that supports the A64 instruction set.
- **x64** compiles your assembly to be run by the 64-bit CLR on a computer that supports the AMD64 or EM64T instruction set.
- **x86** compiles your assembly to be run by the 32-bit, x86-compatible CLR.
- **Itanium** compiles your assembly to be run by the 64-bit CLR on a computer with an Itanium processor.

On a 64-bit Windows operating system:

- Assemblies compiled with **x86** execute on the 32-bit CLR running under WOW64.
- A DLL compiled with the **anycpu** executes on the same CLR as the process into which it's loaded.
- Executables that are compiled with the **anycpu** execute on the 64-bit CLR.
- Executables compiled with **anycpu32bitpreferred** execute on the 32-bit CLR.

The **anycpu32bitpreferred** setting is valid only for executable (.EXE) files, and it requires .NET Framework 4.5 or later. For more information about developing an application to run on a Windows 64-bit operating system, see [64-bit Applications](#).

You set the **PlatformTarget** option from **Build** properties page for your project in Visual Studio.

The behavior of **anycpu** has some additional nuances on .NET Core and .NET 5 and later releases. When you set **anycpu**, publish your app and execute it with either the **x86** `dotnet.exe` or the **x64** `dotnet.exe`. For self-contained apps, the `dotnet publish` step packages the executable for the configure RID.

ProduceReferenceAssembly

The **ProduceReferenceAssembly** option specifies a file path where the reference assembly should be output. It translates to `metadataPeStream` in the Emit API. The `filepath` specifies the path for the reference assembly. It should generally match that of the primary assembly. The recommended convention (used by MSBuild) is to place the reference assembly in a "ref/" subfolder relative to the primary assembly.

```
<ProduceReferenceAssembly>filepath</ProduceReferenceAssembly>
```

Reference assemblies are a special type of assembly that contains only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools. Reference assemblies exclude all member implementations and declarations of private members. Those members have no observable impact on their API

contract. For more information, see [Reference assemblies](#) in the .NET Guide.

The `ProduceReferenceAssembly` and `ProduceOnlyReferenceAssembly` options are mutually exclusive.

TargetType

The `TargetType` compiler option can be specified in one of the following forms:

- **library**: to create a code library. **library** is the default value.
- **exe**: to create an .exe file.
- **module** to create a module.
- **winexe** to create a Windows program.
- **winmdobj** to create an intermediate `.winmdobj` file.
- **appcontainerexe** to create an .exe file for Windows 8.x Store apps.

NOTE

For .NET Framework targets, unless you specify **module**, this option causes a .NET Framework assembly manifest to be placed in an output file. For more information, see [Assemblies in .NET](#) and [Common Attributes](#).

```
<TargetType>library</TargetType>
```

The compiler creates only one assembly manifest per compilation. Information about all files in a compilation is placed in the assembly manifest. When producing multiple output files at the command line, only one assembly manifest can be created and it must go into the first output file specified on the command line.

If you create an assembly, you can indicate that all or part of your code is CLS-compliant with the [CLSCompliantAttribute](#) attribute.

library

The **library** option causes the compiler to create a dynamic-link library (DLL) rather than an executable file (EXE). The DLL will be created with the `.dll` extension. Unless otherwise specified with the [OutputAssembly](#) option, the output file name takes the name of the first input file. When building a `.dll` file, a `Main` method isn't required.

exe

The **exe** option causes the compiler to create an executable (EXE), console application. The executable file will be created with the `.exe` extension. Use **winexe** to create a Windows program executable. Unless otherwise specified with the [OutputAssembly](#) option, the output file name takes the name of the input file that contains the entry point (`Main` method or top-level statements). One and only one entry point is required in the source code files that are compiled into an .exe file. The [StartupObject](#) compiler option lets you specify which class contains the `Main` method, in cases where your code has more than one class with a `Main` method.

module

This option causes the compiler to not generate an assembly manifest. By default, the output file created by compiling with this option will have an extension of `.netmodule`. A file that doesn't have an assembly manifest cannot be loaded by the .NET runtime. However, such a file can be incorporated into the assembly manifest of an assembly with [AddModules](#). If more than one module is created in a single compilation, [internal](#) types in one module will be available to other modules in the compilation. When code in one module references [internal](#) types in another module, then both modules must be incorporated into an assembly manifest, with [AddModules](#). Creating a module isn't supported in the Visual Studio development environment.

winexe

The **winexe** option causes the compiler to create an executable (EXE), Windows program. The executable file will be created with the .exe extension. A Windows program is one that provides a user interface from either the .NET library or with the Windows APIs. Use **exe** to create a console application. Unless otherwise specified with the **OutputAssembly** option, the output file name takes the name of the input file that contains the **Main** method. One and only one **Main** method is required in the source code files that are compiled into an .exe file. The **StartupObject** option lets you specify which class contains the **Main** method, in cases where your code has more than one class with a **Main** method.

winmdobj

If you use the **winmdobj** option, the compiler creates an intermediate *.winmdobj* file that you can convert to a Windows Runtime binary (*.winmd*) file. The *.winmd* file can then be consumed by JavaScript and C++ programs, in addition to managed language programs.

The **winmdobj** setting signals to the compiler that an intermediate module is required. The *.winmdobj* file can then be fed through the **WinMDExp** export tool to produce a Windows metadata (*.winmd*) file. The *.winmd* file contains both the code from the original library and the WinMD metadata that is used by JavaScript or C++ and by the Windows Runtime. The output of a file that's compiled by using the **winmdobj** compiler option is used only as input for the **WinMDExp** export tool. The *.winmdobj* file itself isn't referenced directly. Unless you use the **OutputAssembly** option, the output file name takes the name of the first input file. A **Main** method isn't required.

appcontainerexe

If you use the **appcontainerexe** compiler option, the compiler creates a Windows executable (.exe) file that must be run in an app container. This option is equivalent to **-target:winexe** but is designed for Windows 8.x Store apps.

To require the app to run in an app container, this option sets a bit in the **Portable Executable** (PE) file. When that bit is set, an error occurs if the **CreateProcess** method tries to launch the executable file outside an app container. Unless you use the **OutputAssembly** option, the output file name takes the name of the input file that contains the **Main** method.

C# Compiler Options that specify inputs

3/15/2021 • 5 minutes to read • [Edit Online](#)

The following options control compiler inputs. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **References** / `-reference` or `-references` : Reference metadata from the specified assembly file or files.
- **AddModules** / `-addmodule` : Add a module (created with `target:module`) to this assembly.)
- **EmbedInteropTypes** / `-link` : Embed metadata from the specified interop assembly files.

References

The **References** option causes the compiler to import [public](#) type information in the specified file into the current project, enabling you to reference metadata from the specified assembly files.

```
<Reference Include="filename" />
```

`filename` is the name of a file that contains an assembly manifest. To import more than one file, include a separate **Reference** element for each file. You can define an alias as a child element of the **Reference** element:

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

In the previous example, `LS` is the valid C# identifier that represents a root namespace that will contain all namespaces in the assembly *filename.dll*. The files you import must contain a manifest. Use [AdditionalLibPaths](#) to specify the directory in which one or more of your assembly references is located. The [AdditionalLibPaths](#) topic also discusses the directories in which the compiler searches for assemblies. In order for the compiler to recognize a type in an assembly, and not in a module, it needs to be forced to resolve the type, which you can do by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler: for example, if you inherit from a type in an assembly, the type name will then be recognized by the compiler. Sometimes it is necessary to reference two different versions of the same component from within one assembly. To do this, use the **Aliases** element on the **References** element for each file to distinguish between the two files. This alias will be used as a qualifier for the component name, and will resolve to the component in one of the files.

NOTE

In Visual Studio, use the **Add Reference** command. For more information, see [How to: Add or Remove References By Using the Reference Manager](#).

AddModules

This option adds a module that was created with the `<TargetType>module</TargetType>` switch to the current compilation:

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Where `file`, `file2` are output files that contain metadata. The file can't contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon. All modules added with **AddModules** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module isn't in the application directory at run time, you'll get a [TypeLoadException](#). `file` can't contain an assembly. For example, if the output file was created with [TargetType](#) option of **module**, its metadata can be imported with **AddModules**.

If the output file was created with a [TargetType](#) option other than **module**, its metadata cannot be imported with **AddModules** but can be imported with the [References](#) option.

EmbedInteropTypes

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Where `file1;file2;file3` is a semicolon-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks. The **EmbedInteropTypes** option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see [Walkthrough: Embedding Types from Managed Assemblies](#).

Using the **EmbedInteropTypes** option is especially useful when you're working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The **EmbedInteropTypes** option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types. The **EmbedInteropTypes** option embeds only interfaces, structures, and delegates. Embedding COM classes isn't supported.

NOTE

When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the `CoClass` causes an error.

Like the [References](#) compiler option, the **EmbedInteropTypes** compiler option uses the `Csc.rsp` response file, which references frequently used .NET assemblies. Use the [NoConfig](#) compiler option if you don't want the compiler to use the `Csc.rsp` file.

```
// The following code causes an error if ISampleInterface is an embedded interop type.  
ISampleInterface<SampleType> sample;
```

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction doesn't apply to interfaces. For example, consider the [Range](#) interface that is defined in the [Microsoft.Office.Interop.Excel](#) assembly. If a library embeds interop types from the [Microsoft.Office.Interop.Excel](#) assembly and exposes a method that returns a generic type that has a parameter whose type is the [Range](#) interface, that method must return a generic interface, as shown in the following code example.

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Microsoft.Office.Interop.Excel;  
  
public class Utility  
{  
    // The following code causes an error when called by a client assembly.  
    public List<Range> GetRange1()  
    {  
        return null;  
    }  
  
    // The following code is valid for calls from a client assembly.  
    public IList<Range> GetRange2()  
    {  
        return null;  
    }  
}
```

In the following example, client code can call the method that returns the [IList](#) generic interface without error.

```
public class Client  
{  
    public void Main()  
    {  
        Utility util = new Utility();  
  
        // The following code causes an error.  
        List<Range> rangeList1 = util.GetRange1();  
  
        // The following code is valid.  
        List<Range> rangeList2 = (List<Range>)util.GetRange2();  
    }  
}
```

C# Compiler Options to report errors and warnings

3/15/2021 • 3 minutes to read • [Edit Online](#)

The following options control how the compiler reports errors and warnings. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **WarningLevel** / `-warn` : Set warning level.
- **TreatWarningsAsErrors** / `-warnaserror` : Treat all warnings as errors
- **WarningsAsErrors** / `-warnaserror` : Treat one or more warnings as errors
- **WarningsNotAsErrors** / `-warnaserror` : Treat one or more warnings not as errors
- **DisabledWarnings** / `-nowarn` : Set a list of disabled warnings.
- **CodeAnalysisRuleSet** / `-ruleset` : Specify a ruleset file that disables specific diagnostics.
- **ErrorLog** / `-errorlog` : Specify a file to log all compiler and analyzer diagnostics.
- **ReportAnalyzer** / `-reportanalyzer` : Report additional analyzer information, such as execution time.

WarningLevel

The **WarningLevel** option specifies the warning level for the compiler to display.

```
<WarningLevel>3</WarningLevel>
```

The element value is the warning level you want displayed for the compilation: Lower numbers show only high severity warnings. Higher numbers show more warnings. The value must be zero or a positive integer:

WARNING LEVEL	MEANING
0	Turns off emission of all warning messages.
1	Displays severe warning messages.
2	Displays level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays level 2 warnings plus certain, less-severe warnings, such as warnings about expressions that always evaluate to <code>true</code> or <code>false</code> .
4 (the default)	Displays all level 3 warnings plus informational warnings.
5	Displays level 4 warnings plus additional warnings from the compiler shipped with C# 9.0.
Greater than 5	Any value greater than 5 will be treated as 5. To make sure you always have all warnings if the compiler is updated with new warning levels, put an arbitrary large value (for example, 9999).

To get information about an error or warning, you can look up the error code in the Help Index. For other ways to get information about an error or warning, see [C# Compiler Errors](#). Use **TreatWarningsAsErrors** to treat all

warnings as errors. Use [DisabledWarnings](#) to disable certain warnings.

TreatWarningsAsErrors

The **TreatWarningsAsErrors** option treats all warnings as errors. You can also use the **TreatWarningsAsErrors** to set only some warnings as errors. If you turn on **TreatWarningsAsErrors**, you can use **TreatWarningsAsErrors** to list warnings that shouldn't be treated as errors.

```
<TreatWarningsAsErrors></TreatWarningsAsErrors>
```

All warning messages are instead reported as errors. The build process halts (no output files are built). By default, **TreatWarningsAsErrors** isn't in effect, which means warnings don't prevent the generation of an output file. Optionally, if you want only a few specific warnings to be treated as errors, you may specify a comma-separated list of warning numbers to treat as errors. The set of all nullability warnings can be specified with the **Nullable** shorthand. Use [WarningLevel](#) to specify the level of warnings that you want the compiler to display. Use [DisabledWarnings](#) to disable certain warnings.

WarningsAsErrors and WarningsNotAsErrors

The **WarningsAsErrors** and **WarningsNotAsErrors** options override the **TreatWarningsAsErrors** option for a list of warnings.

Enable warnings 0219 and 0168 as errors:

```
<WarningsAsErrors>0219,0168</WarningsAsErrors>
```

Disable the same warnings as errors:

```
<WarningsNotAsErrors>0219,0168</WarningsNotAsErrors>
```

You use **WarningsAsErrors** to configure a set of warnings as errors. Use **WarningsNotAsErrors** to configure a set of warnings that should not be errors when you've set all warnings as errors.

DisabledWarnings

The **DisabledWarnings** option lets you suppress the compiler from displaying one or more warnings. Separate multiple warning numbers with a comma.

```
<DisabledWarnings>number1, number2</DisabledWarnings>
```

`number1, number2` Warning number(s) that you want the compiler to suppress. You specify the numeric part of the warning identifier. For example, if you want to suppress *CS0028*, you could specify

`<DisabledWarnings>28</DisabledWarnings>`. The compiler silently ignores warning numbers passed to **DisabledWarnings** that were valid in previous releases, but that have been removed. For example, *CS0679* was valid in the compiler in Visual Studio .NET 2002 but was removed later.

The following warnings cannot be suppressed by the **DisabledWarnings** option:

- Compiler Warning (level 1) CS2002
- Compiler Warning (level 1) CS2023
- Compiler Warning (level 1) CS2029

CodeAnalysisRuleSet

Specify a ruleset file that configures specific diagnostics.

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

Where `MyConfiguration.ruleset` is the path to the ruleset file. For more information on using rule sets, see the article in the [Visual Studio documentation on Rule sets](#).

ErrorLog

Specify a file to log all compiler and analyzer diagnostics.

```
<ErrorLog>MyConfiguration.ruleset</ErrorLog>
```

The **ErrorLog** option causes the compiler to output a [Static Analysis Results Interchange Format \(SARIF\) log](#). SARIF logs are typically read by tools that analyze the results from compiler and analyzer diagnostics.

ReportAnalyzer

Report additional analyzer information, such as execution time.

```
<ReportAnalyzer>true</ReportAnalyzer>
```

The **ReportAnalyzer** option causes the compiler to emit extra MSBuild log information that details the performance characteristics of analyzers in the build. It's typically used by analyzer authors as part of validating the analyzer.

C# Compiler Options that control code generation

3/31/2021 • 4 minutes to read • [Edit Online](#)

The following options control code generation by the compiler. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **DebugType** / `-debug` : Emit (or don't emit) debugging information.
- **Optimize** / `-optimize` : Enable optimizations.
- **Deterministic** / `-deterministic` : Produce byte-for-byte equivalent output from the same input source.
- **ProduceOnlyReferenceAssembly** / `-refonly` : Produce a reference assembly, instead of a full assembly, as the primary output.

DebugType

The **DebugType** option causes the compiler to generate debugging information and place it in the output file or files. Debugging information is added by default for the *Debug* build configuration. It is off by default for the *Release* build configuration.

```
<DebugType>pdbonly</DebugType>
```

For all compiler versions starting with C# 6.0, there is no difference between *pdbonly* and *full*. Choose *pdbonly*. To change the location of the *.pdb* file, see [PdbFile](#).

The following values are valid:

VALUE	MEANING
<code>full</code>	Emit debugging information to <i>.pdb</i> file using default format for the current platform: Windows : A Windows pdb file. Linux/macOS : A Portable PDB file.
<code>pdbonly</code>	Same as <code>full</code> . See the note below for more information.
<code>portable</code>	Emit debugging information to <i>.pdb</i> file using cross-platform Portable PDB format.
<code>embedded</code>	Emit debugging information into the <i>.dll/.exe</i> itself (<i>.pdb</i> file is not produced) using Portable PDB format.

IMPORTANT

The following information applies only to compilers older than C# 6.0. The value of this element can be either `full` or `pdbonly`. The `full` argument, which is in effect if you don't specify `pdbonly`, enables attaching a debugger to the running program. Specifying `pdbonly` allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger. Use this option to create debug builds. If you use `Full`, be aware that there's some impact on the speed and size of JIT optimized code and a small impact on code quality with `full`. We recommend `pdbonly` or no PDB for generating release code. One difference between `pdbonly` and `full` is that with `full` the compiler emits a [DebuggableAttribute](#), which is used to tell the JIT compiler that debug information is available. Therefore, you will get an error if your code contains the [DebuggableAttribute](#) set to false if you use `full`. For more information on how to configure the debug performance of an application, see [Making an Image Easier to Debug](#).

Optimize

The **Optimize** option enables or disables optimizations performed by the compiler to make your output file smaller, faster, and more efficient. The *Optimize* option is enabled by default for a *Release* build configuration. It is off by default for a *Debug* build configuration.

```
<Optimize>true</Optimize>
```

You set the **Optimize** option from **Build** properties page for your project in Visual Studio.

Optimize also tells the common language runtime to optimize code at runtime. By default, optimizations are disabled. Specify **Optimize+** to enable optimizations. When building a module to be used by an assembly, use the same **Optimize** settings as used by the assembly. It's possible to combine the **Optimize** and **Debug** options.

Deterministic

Causes the compiler to produce an assembly whose byte-for-byte output is identical across compilations for identical inputs.

```
<Deterministic>true</Deterministic>
```

By default, compiler output from a given set of inputs is unique, since the compiler adds a timestamp and an MVID that is generated from random numbers. You use the `<Deterministic>` option to produce a *deterministic assembly*, one whose binary content is identical across compilations as long as the input remains the same. In such a build, the timestamp and MVID fields will be replaced with values derived from a hash of all the compilation inputs. The compiler considers the following inputs that affect determinism:

- The sequence of command-line parameters.
- The contents of the compiler's .rsp response file.
- The precise version of the compiler used, and its referenced assemblies.
- The current directory path.
- The binary contents of all files explicitly passed to the compiler either directly or indirectly, including:
 - Source files
 - Referenced assemblies
 - Referenced modules
 - Resources
 - The strong name key file
 - @ response files

- Analyzers
- Rulesets
- Other files that may be used by analyzers
- The current culture (for the language in which diagnostics and exception messages are produced).
- The default encoding (or the current code page) if the encoding isn't specified.
- The existence, non-existence, and contents of files on the compiler's search paths (specified, for example, by `-lib` or `-recurse`).
- The Common Language Runtime (CLR) platform on which the compiler is run.
- The value of `%LIBPATH%`, which can affect analyzer dependency loading.

Deterministic compilation can be used for establishing whether a binary is compiled from a trusted source. Deterministic output can be useful when the source is publicly available. It can also determine whether build steps that are dependent on changes to binary used in the build process.

ProduceOnlyReferenceAssembly

The **ProduceOnlyReferenceAssembly** option indicates that a reference assembly should be output instead of an implementation assembly, as the primary output. The **ProduceOnlyReferenceAssembly** parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Reference assemblies are a special type of assembly. Reference assemblies contain only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools, but exclude all member implementations and declarations of private members that have no observable impact on their API contract. For more information, see [Reference assemblies](#).

The **ProduceOnlyReferenceAssembly** and **ProduceReferenceAssembly** options are mutually exclusive.

C# Compiler Options for security options

3/15/2021 • 4 minutes to read • [Edit Online](#)

The following options control compiler security options. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **PublicSign** / `-publicsign` : Publicly sign the assembly.
- **DelaySign** / `-delaysign` : Delay-sign the assembly using only the public portion of the strong name key.
- **KeyFile** / `-keyfile` : Specify a strong name key file.
- **KeyContainer** / `-keycontainer` : Specify a strong name key container.
- **HighEntropyVA** / `-highentropyva` : Enable high-entropy Address Space Layout Randomization (ASLR)

PublicSign

This option causes the compiler to apply a public key but doesn't actually sign the assembly. The **PublicSign** option also sets a bit in the assembly that tells the runtime that the file is signed.

```
<PublicSign>true</PublicSign>
```

The **PublicSign** option requires the use of the **KeyFile** or **KeyContainer** option. The **keyFile** and **KeyContainer** options specify the public key. The **PublicSign** and **PublicSign** options are mutually exclusive. Sometimes called "fake sign" or "OSS sign", public signing includes the public key in an output assembly and sets the "signed" flag. Public signing doesn't actually sign the assembly with a private key. Developers use public sign for open-source projects. People build assemblies that are compatible with the released "fully signed" assemblies when they don't have access to the private key used to sign the assemblies. Since few consumers actually need to check if the assembly is fully signed, these publicly built assemblies are useable in almost every scenario where the fully signed one would be used.

DelaySign

This option causes the compiler to reserve space in the output file so that a digital signature can be added later.

```
<DelaySign>true</DelaySign>
```

Use **DelaySign** - if you want a fully signed assembly. Use **DelaySign** if you only want to place the public key in the assembly. The **DelaySign** option has no effect unless used with **KeyFile** or **KeyContainer**. The **KeyContainer** and **PublicSign** options are mutually exclusive. When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. That operation creates a digital signature that is stored in the file that contains the manifest. When an assembly is delay signed, the compiler doesn't compute and store the signature. Instead, the compiler but reserves space in the file so the signature can be added later.

Using **DelaySign** allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly using the [Assembly Linker](#) utility. For more information, see [Creating and Using Strong-Named Assemblies](#) and [Delay Signing an Assembly](#).

KeyFile

Specifies the filename containing the cryptographic key.

```
<KeyFile>filename</KeyFile>
```

`file` is the name of the file containing the strong name key. When this option is used, the compiler inserts the public key from the specified file into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. If you compile with [-target:module](#), the name of the key file is held in the module and incorporated into the assembly created when you compile an assembly with [AddModules](#). You can also pass your encryption information to the compiler with [KeyContainer](#). Use [DelaySign](#) if you want a partially signed assembly. In case both [KeyFile](#) and [KeyContainer](#) are specified in the same compilation, the compiler will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler doesn't find the key container, it will try the file specified with [KeyFile](#). If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container. On the next compilation, the key container will be valid. A key file might contain only the public key. For more information, see [Creating and Using Strong-Named Assemblies](#) and [Delay Signing an Assembly](#).

KeyContainer

Specifies the name of the cryptographic key container.

```
<KeyContainer>container</KeyContainer>
```

`container` is the name of the strong name key container. When the [KeyContainer](#) option is used, the compiler creates a sharable component. The compiler inserts a public key from the specified container into the assembly manifest and signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. `sn -i` installs the key pair into a container. This option isn't supported when the compiler runs on CoreCLR. To sign an assembly when building on CoreCLR, use the [KeyFile](#) option. If you compile with [TargetType](#), the name of the key file is held in the module and incorporated into the assembly when you compile this module into an assembly with [AddModules](#). You can also specify this option as a custom attribute ([System.Reflection.AssemblyKeyNameAttribute](#)) in the source code for any Microsoft intermediate language (MSIL) module. You can also pass your encryption information to the compiler with [KeyFile](#). Use [DelaySign](#) to add the public key to the assembly manifest but signing the assembly until it has been tested. For more information, see [Creating and Using Strong-Named Assemblies](#) and [Delay Signing an Assembly](#).

HighEntropyVA

The [HighEntropyVA](#) compiler option tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).

```
<HighEntropyVA>true</HighEntropyVA>
```

This option specifies that a 64-bit executable or an executable that is marked by the [PlatformTarget](#) compiler option supports a high entropy virtual address space. The option is disabled by default. Use [HighEntropyVA](#) to enable it.

The [HighEntropyVA](#) option enables compatible versions of the Windows kernel to use higher degrees of entropy when randomizing the address space layout of a process as part of ASLR. Using higher degrees of entropy means a larger number of addresses can be allocated to memory regions such as stacks and heaps. As a result, it's more difficult to guess the location of a particular memory region. The [HighEntropyVA](#) compiler option requires the target executable and any modules that it depends on to handle pointer values larger than 4

gigabytes (GB) when they're running as a 64-bit process.

C# Compiler Options that specify resources

3/15/2021 • 5 minutes to read • [Edit Online](#)

The following options control how the C# compiler creates or imports Win32 resources. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **Win32Resource** / `-win32res` : Specify a Win32 resource file (.res).
- **Win32Icon** / `-win32icon` : Reference metadata from the specified assembly file or files.
- **Win32Manifest** / `-win32manifest` : Specify a Win32 manifest file (xml).
- **NoWin32Manifest** / `-nowin32manifest` : Don't include the default Win32 manifest.
- **Resources** / `-resource` : Embed the specified resource (Short form: /res).
- **LinkResources** / `-linkresources` : Link the specified resource to this assembly.

Win32Resource

The **Win32Resource** option inserts a Win32 resource in the output file.

```
<Win32Resource>filename</Win32Resource>
```

`filename` is the resource file that you want to add to your output file. A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the File Explorer. If you don't specify this option, the compiler will generate version information based on the assembly version.

Win32Icon

The **Win32Icon** option inserts an .ico file in the output file, which gives the output file the desired appearance in the File Explorer.

```
<Win32Icon>filename</Win32Icon>
```

`filename` is the .ico file that you want to add to your output file. An .ico file can be created with the [Resource Compiler](#). The Resource Compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file.

Win32Manifest

Use the **Win32Manifest** option to specify a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` is the name and location of the custom manifest file. By default, the C# compiler embeds an application manifest that specifies a requested execution level of "asInvoker". It creates the manifest in the same folder in which the executable is built. If you want to supply a custom manifest, for example to specify a requested execution level of "highestAvailable" or "requireAdministrator," use this option to specify the name of the file.

NOTE

This option and the **Win32Resources** option are mutually exclusive. If you try to use both options in the same command line you will get a build error.

An application that has no application manifest that specifies a requested execution level will be subject to file and registry virtualization under the User Account Control feature in Windows. For more information, see [User Account Control](#).

Your application will be subject to virtualization if either of these conditions is true:

- You use the **NoWin32Manifest** option and you don't provide a manifest in a later build step or as part of a Windows Resource (.res) file by using the **Win32Resource** option.
- You provide a custom manifest that doesn't specify a requested execution level.

Visual Studio creates a default *.manifest* file and stores it in the debug and release directories alongside the executable file. You can add a custom manifest by creating one in any text editor and then adding the file to the project. Or, you can right-click the **Project** icon in **Solution Explorer**, select **Add New Item**, and then select **Application Manifest File**. After you've added your new or existing manifest file, it will appear in the **Manifest** drop down list. For more information, see [Application Page, Project Designer \(C#\)](#).

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the **NoWin32Manifest** option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista.

NoWin32Manifest

Use the **NoWin32Manifest** option to instruct the compiler not to embed any application manifest into the executable file.

```
<NoWin32Manifest />
```

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step.

In Visual Studio, set this option in the **Application Property** page by selecting the **Create Application Without a Manifest** option in the **Manifest** drop down list. For more information, see [Application Page, Project Designer \(C#\)](#).

Resources

Embeds the specified resource into the output file.

```
<Resources Include="filename">
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` is the .NET resource file that you want to embed in the output file. `identifier` (optional) is the logical name for the resource; the name that is used to load the resource. The default is the name of the file.

`accessibility-modifier` (optional) is the accessibility of the resource: public or private. The default is public. By default, resources are public in the assembly when they're created by using the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other accessibility other than `public` or

`private` is allowed. If `filename` is a .NET resource file created, for example, by [Resgen.exe](#) or in the development environment, it can be accessed with members in the [System.Resources](#) namespace. For more information, see [System.Resources.ResourceManager](#). For all other resources, use the `GetManifestResource` methods in the [Assembly](#) class to access the resource at run time. The order of the resources in the output file is determined from the order specified in the project file.

LinkResources

Creates a link to a .NET resource in the output file. The resource file isn't added to the output file.

LinkResources differs from the **Resource** option, which does embed a resource file in the output file.

```
<LinkResources Include="filename">
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` is the .NET resource file to which you want to link from the assembly. `identifier` (optional) is the logical name for the resource; the name that is used to load the resource. The default is the name of the file. `accessibility-modifier` (optional) is the accessibility of the resource: public or private. The default is public. By default, linked resources are public in the assembly when they're created with the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other modifier other than `public` or `private` is allowed. If `filename` is a .NET resource file created, for example, by [Resgen.exe](#) or in the development environment, it can be accessed with members in the [System.Resources](#) namespace. For more information, see [System.Resources.ResourceManager](#). For all other resources, use the `GetManifestResource` methods in the [Assembly](#) class to access the resource at run time. The file specified in `filename` can be any format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly. You can do the same thing in the Assembly Linker. For more information, see [Al.exe \(Assembly Linker\)](#) and [Working with Assemblies and the Global Assembly Cache](#).

Miscellaneous C# Compiler Options

3/15/2021 • 2 minutes to read • [Edit Online](#)

The following options control miscellaneous compiler behavior. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **ResponseFiles** / `-@:` Read response file for more options.
- **NoLogo** / `-nologo`: Suppress compiler copyright message.
- **NoConfig** / `-noconfig`: Don't auto include *CSC.RSP* file.

ResponseFiles

The **ResponseFiles** option lets you specify a file that contains compiler options and source code files to compile.

```
<ResponseFiles>response_file</ResponseFiles>
```

The `response_file` specifies the file that lists compiler options or source code files to compile. The compiler options and source code files will be processed by the compiler as if they had been specified on the command line. To specify more than one response file in a compilation, specify multiple response file options. In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (can't span multiple lines). Response files can have comments that begin with the # symbol. Specifying compiler options from within a response file is just like issuing those commands on the command line. The compiler processes the command options as they're read. Command-line arguments can override previously listed options in response files. Conversely, options in a response file will override options listed previously on the command line or in other response files. C# provides the *csc.rsp* file, which is located in the same directory as the *csc.exe* file. For more information about the response file format, see [NoConfig](#). This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically. The following are a few lines from a sample response file:

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

The **NoLogo** option suppresses display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

```
<NoLogo>true</NoLogo>
```

NoConfig

The **NoConfig** option tells the compiler not to compile with the *csc.rsp* file.

```
<NoConfig>true</NoConfig>
```

The *csc.rsp* file references all the assemblies shipped with .NET Framework. The actual references that the Visual

Studio .NET development environment includes depend on the project type. You can modify the *csc.rsp* file and specify additional compiler options that should be included in every compilation. If you don't want the compiler to look for and use the settings in the *csc.rsp* file, specify **NoConfig**. This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Advanced C# compiler options

3/25/2021 • 10 minutes to read • [Edit Online](#)

The following options support advanced scenarios. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **MainEntryPoint**, **StartupObject** / `-main`: Specify the type that contains the entry point.
- **PdbFile** / `-pdb`: Specify debug information file name.
- **PathMap** / `-pathmap`: Specify a mapping for source path names output by the compiler.
- **ApplicationConfiguration** / `-appconfig`: Specify an application configuration file containing assembly binding settings.
- **AdditionalLibPaths** / `-lib`: Specify additional directories to search in for references.
- **GenerateFullPaths** / `-fullpath`: Compiler generates fully qualified paths.
- **PreferredUILang** / `-preferreduilang`: Specify the preferred output language name.
- **BaseAddress** / `-baseaddress`: Specify the base address for the library to be built.
- **ChecksumAlgorithm** / `-checksumalgorithm`: Specify algorithm for calculating source file checksum stored in PDB.
- **CodePage** / `-codepage`: Specify the codepage to use when opening source files.
- **Utf8Output** / `-utf8output`: Output compiler messages in UTF-8 encoding.
- **FileAlignment** / `-filealign`: Specify the alignment used for output file sections.
- **ErrorEndLocation** / `-errorendlocation`: Output line and column of the end location of each error.
- **NoStandardLib** / `-nostdlib`: Don't reference standard library *mscorlib.dll*.
- **SubsystemVersion** / `-subsystemversion`: Specify subsystem version of this assembly.
- **ModuleAssemblyName** / `-moduleassemblyname`: Name of the assembly that this module will be a part of.

MainEntryPoint or StartupObject

This option specifies the class that contains the entry point to the program, if more than one class contains a `Main` method.

```
<StartupObject>MyNamespace.Program</StartupObject>
```

or

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

Where `Program` is the type that contains the `Main` method. The provided class name must be fully qualified; it must include the full namespace containing the class, followed by the class name. For example, when the `Main` method is located inside the `Program` class in the `MyApplication.Core` namespace, the compiler option has to be `-main:MyApplication.Core.Program`. If your compilation includes more than one type with a `Main` method, you can specify which type contains the `Main` method.

NOTE

This option can't be used for a project that includes [top-level statements](#), even if that project contains one or more `Main` methods.

PdbFile

The **PdbFile** compiler option specifies the name and location of the debug symbols file. The `filename` value specifies the name and location of the debug symbols file.

```
<PdbFile>filename</PdbFile>
```

When you specify [DebugType](#), the compiler will create a `.pdb` file in the same directory where the compiler will create the output file (.exe or .dll). The `.pdb` file has the same base file name as the name of the output file. **PdbFile** allows you to specify a non-default file name and location for the `.pdb` file. This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically.

PathMap

The **PathMap** compiler option specifies how to map physical paths to source path names output by the compiler. This option maps each physical path on the machine where the compiler runs to a corresponding path that should be written in the output files. In the following example, `path1` is the full path to the source files in the current environment, and `sourcePath1` is the source path substituted for `path1` in any output files. To specify multiple mapped source paths, separate each with a semicolon.

```
<PathMap>path1=sourcePath1;path2=sourcePath2</PathMap>
```

The compiler writes the source path into its output for the following reasons:

1. The source path is substituted for an argument when the [CallerFilePathAttribute](#) is applied to an optional parameter.
2. The source path is embedded in a PDB file.
3. The path of the PDB file is embedded into a PE (portable executable) file.

ApplicationConfiguration

The **ApplicationConfiguration** compiler option enables a C# application to specify the location of an assembly's application configuration (app.config) file to the common language runtime (CLR) at assembly binding time.

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

Where `file` is the application configuration file that contains assembly binding settings. One use of **ApplicationConfiguration** is advanced scenarios in which an assembly has to reference both the .NET Framework version and the .NET Framework for Silverlight version of a particular reference assembly at the same time. For example, a XAML designer written in Windows Presentation Foundation (WPF) might have to reference both the WPF Desktop, for the designer's user interface, and the subset of WPF that is included with Silverlight. The same designer assembly has to access both assemblies. By default, the separate references cause a compiler error, because assembly binding sees the two assemblies as equivalent. The **ApplicationConfiguration** compiler option enables you to specify the location of an app.config file that

disables the default behavior by using a `<supportPortability>` tag, as shown in the following example.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

The compiler passes the location of the file to the CLR's assembly-binding logic.

NOTE

To use the app.config file that is already set in the project, add property tag `<UseAppConfigForCompiler>` to the `.csproj` file and set its value to `true`. To specify a different app.config file, add property tag `< AppConfigForCompiler>` and set its value to the location of the file.

The following example shows an app.config file that enables an application to have references to both the .NET Framework implementation and the .NET Framework for Silverlight implementation of any .NET Framework assembly that exists in both implementations. The **ApplicationConfiguration** compiler option specifies the location of this app.config file.

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

The **AdditionalLibPaths** option specifies the location of assemblies referenced with the [References](#) option.

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

Where `dir1` is a directory for the compiler to look in if a referenced assembly isn't found in the current working directory (the directory from which you're invoking the compiler) or in the common language runtime's system directory. `dir2` is one or more additional directories to search in for assembly references. Separate directory names with a comma, and without white space between them. The compiler searches for assembly references that aren't fully qualified in the following order:

1. Current working directory.
2. The common language runtime system directory.
3. Directories specified by **AdditionalLibPaths**.
4. Directories specified by the LIB environment variable.

Use **Reference** to specify an assembly reference. **AdditionalLibPaths** is additive. Specifying it more than once appends to any prior values. Since the path to the dependent assembly isn't specified in the assembly manifest, the application will find and use the assembly in the global assembly cache. The compiler referencing the assembly doesn't imply the common language runtime can find and load the assembly at runtime. See [How the Runtime Locates Assemblies](#) for details on how the runtime searches for referenced assemblies.

GenerateFullPaths

The **GenerateFullPaths** option causes the compiler to specify the full path to the file when listing compilation errors and warnings.

```
<GenerateFullPaths>true</GenerateFullPaths>
```

By default, errors and warnings that result from compilation specify the name of the file in which an error was found. The **GenerateFullPaths** option causes the compiler to specify the full path to the file. This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

PreferredUILang

By using the **PreferredUILang** compiler option, you can specify the language in which the C# compiler displays output, such as error messages.

```
<PreferredUILang>language</PreferredUILang>
```

Where `language` is the [language name](#) of the language to use for compiler output. You can use the **PreferredUILang** compiler option to specify the language that you want the C# compiler to use for error messages and other command-line output. If the language pack for the language isn't installed, the language setting of the operating system is used instead.

BaseAddress

The **BaseAddress** option lets you specify the preferred base address at which to load a DLL. For more information about when and why to use this option, see [Larry Osterman's WebLog](#).

```
<BaseAddress>address</BaseAddress>
```

Where `address` is the base address for the DLL. This address can be specified as a decimal, hexadecimal, or octal number. The default base address for a DLL is set by the .NET common language runtime. The lower-order word in this address will be rounded. For example, if you specify `0x11110001`, it will be rounded to `0x11110000`. To complete the signing process for a DLL, use SN.EXE with the -R option.

ChecksumAlgorithm

This option controls the checksum algorithm we use to encode source files in the PDB.

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

The `algorithm` must be either `SHA1` (default) or `SHA256`.

CodePage

This option specifies which codepage to use during compilation if the required page isn't the current default codepage for the system.

```
<CodePage>id</CodePage>
```

Where `id` is the id of the code page to use for all source code files in the compilation. The compiler will first attempt to interpret all source files as UTF-8. If your source code files are in an encoding other than UTF-8 and use characters other than 7-bit ASCII characters, use the **CodePage** option to specify which code page should be used. **CodePage** applies to all source code files in your compilation. See [GetCPIInfo](#) for information on how to find which code pages are supported on your system.

Utf8Output

The **Utf8Output** option displays compiler output using UTF-8 encoding.

```
<Utf8Output>true</Utf8Output>
```

In some international configurations, compiler output cannot correctly be displayed in the console. Use **Utf8Output** and redirect compiler output to a file.

FileAlignment

The **FileAlignment** option lets you specify the size of sections in your output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

```
<FileAlignment>number</FileAlignment>
```

You set the **FileAlignment** option from the **Advanced** page of the **Build** properties for your project in Visual Studio. Each section will be aligned on a boundary that is a multiple of the **FileAlignment** value. There's no fixed default. If **FileAlignment** isn't specified, the common language runtime picks a default at compile time. By specifying the section size, you affect the size of the output file. Modifying section size may be useful for programs that will run on smaller devices. Use **DUMPBIN** to see information about sections in your output file.

ErrorEndLocation

Instructs the compiler to output line and column of the end location of each error.

```
<ErrorEndLocation>filename</ErrorEndLocation>
```

By default, the compiler writes the starting location in source for all errors and warnings. When this option is set to true, the compiler writes both the starting and end location for each error and warning.

NoStandardLib

NoStandardLib prevents the import of mscorelib.dll, which defines the entire System namespace.

```
<NoStandardLib>true</NoStandardLib>
```

Use this option if you want to define or create your own System namespace and objects. If you don't specify **NoStandardLib**, mscorelib.dll is imported into your program (same as specifying

```
<NoStandardLib>false</NoStandardLib> ).
```

SubsystemVersion

Specifies the minimum version of the subsystem on which the executable file runs. Most commonly, this option ensures that the executable file can use security features that aren't available with older versions of Windows.

NOTE

To specify the subsystem itself, use the **TargetType** compiler option.

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

The `major.minor` specify the minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7. Set the value of this option to 6.01, as the table later in this article describes. You specify the values for `major` and `minor` as integers. Leading zeroes in the `minor` version don't change the version, but trailing zeroes do. For example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

The following table lists common subsystem versions of Windows.

WINDOWS VERSION	SUBSYSTEM VERSION
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

The default value of the **SubsystemVersion** compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
 - `-target:appcontainerexe`
 - `-target:winmdobj`
 - `-platform:arm`
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5, and you haven't set any of the compiler options that were specified earlier in this list.
- The default value is 4.00 if none of the previous conditions are true.

ModuleAssemblyName

Specifies the name of an assembly whose non-public types a *.netmodule* can access.

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

ModuleAssemblyName should be used when building a *.netmodule*, and where the following conditions are true:

- The *.netmodule* needs access to non-public types in an existing assembly.
- You know the name of the assembly into which the *.netmodule* will be built.
- The existing assembly has granted friend assembly access to the assembly into which the *.netmodule* will be built.

For more information on building a *.netmodule*, see [TargetType](#) option of **module**. For more information on

friend assemblies, see [Friend Assemblies](#).

C# Compiler Errors

3/15/2021 • 2 minutes to read • [Edit Online](#)

Some C# compiler errors have corresponding topics that explain why the error is generated, and, in some cases, how to fix the error. Use one of the following steps to see whether help is available for a particular error message.

- Find the error number (for example, CS0029) in the [Output Window](#), and then search for it on Microsoft Docs.
- Choose the error number (for example, CS0029) in the [Output Window](#), and then choose the F1 key.
- In the Index, enter the error number in the **Look for** box.

If none of these steps leads to information about your error, go to the end of this page, and send feedback that includes the number or text of the error.

For information about how to configure error and warning options in C#, see [Build Page, Project Designer \(C#\)](#).

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements.

For more information, see [Personalizing the IDE](#).

See also

- [C# Compiler Options](#)
- [Build Page, Project Designer \(C#\)](#)
- [WarningLevel \(C# Compiler Options\)](#)
- [DisabledWarnings \(C# Compiler Options\)](#)

Introduction

3/13/2021 • 55 minutes to read • [Edit Online](#)

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

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

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

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

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

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

Hello world

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

```
using System;

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

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

```
csc hello.cs
```

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

```
Hello, World
```

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

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

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

Program structure

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

The example

```

using System;

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

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

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

    class Entry
    {
        public Entry next;
        public object data;

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

```

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

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

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

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

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

```

using System;
using Acme.Collections;

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

```

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

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

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

```

100
10
1

```

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

Types and variables

There are two kinds of types in C#: *value types* and *reference types*. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other (except in the case of `ref` and `out` parameter variables).

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

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

CATEGORY		DESCRIPTION
Value types	Simple types	Signed integral: <code>sbyte</code> , <code>short</code> , <code>int</code> , <code>long</code>
		Unsigned integral: <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code>

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

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

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

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

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

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

The following table summarizes C#'s numeric types.

CATEGORY	BITS	TYPE	RANGE/PRECISION
Signed integral	8	<code>sbyte</code>	-128...127

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

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

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

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

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

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

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

types.

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

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

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

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

```
using System;

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

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

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

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

TYPE OF VARIABLE	POSSIBLE CONTENTS
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
<code>object</code>	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type

TYPE OF VARIABLE	POSSIBLE CONTENTS
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 of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

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

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

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

CATEGORY	EXPRESSION	DESCRIPTION
Primary	<code>x.m</code>	Member access
	<code>x(...)</code>	Method and delegate invocation
	<code>x[...]</code>	Array and indexer access
	<code>x++</code>	Post-increment
	<code>x--</code>	Post-decrement
	<code>new T(...)</code>	Object and delegate creation
	<code>new T(...){...}</code>	Object creation with initializer
	<code>new {...}</code>	Anonymous object initializer
	<code>new T[...]</code>	Array creation

CATEGORY	EXPRESSION	DESCRIPTION
	<code>typeof(T)</code>	Obtain <code>System.Type</code> object for <code>T</code>
	<code>checked(x)</code>	Evaluate expression in checked context
	<code>unchecked(x)</code>	Evaluate expression in unchecked context
	<code>default(T)</code>	Obtain default value of type <code>T</code>
	<code>delegate {...}</code>	Anonymous function (anonymous method)
Unary	<code>+x</code>	Identity
	<code>-x</code>	Negation
	<code>!x</code>	Logical negation
	<code>~x</code>	Bitwise negation
	<code>++x</code>	Pre-increment
	<code>--x</code>	Pre-decrement
	<code>(T)x</code>	Explicitly convert <code>x</code> to type <code>T</code>
	<code>await x</code>	Asynchronously wait for <code>x</code> to complete
Multiplicative	<code>x * y</code>	Multiplication
	<code>x / y</code>	Division
	<code>x % y</code>	Remainder
Additive	<code>x + y</code>	Addition, string concatenation, delegate combination
	<code>x - y</code>	Subtraction, delegate removal
Shift	<code>x << y</code>	Shift left
	<code>x >> y</code>	Shift right
Relational and type testing	<code>x < y</code>	Less than
	<code>x > y</code>	Greater than
	<code>x <= y</code>	Less than or equal

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

Statements

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

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

Declaration statements are used to declare local variables and constants.

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

expressions.

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

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

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

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

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

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

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

Below are examples of each kind of statement

Local variable declarations

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

Local constant declaration

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

Expression statement

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

`if` statement

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

switch statement

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

while statement

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

do statement

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

for statement

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

foreach statement

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

break statement

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

continue statement

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

goto statement

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

return statement

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

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

yield statement

```

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

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

```

throw and **try** statements

```

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

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

```

checked and **unchecked** statements

```

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

```

lock statement

```

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

```

using statement

```

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

```

Classes and objects

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

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

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

```

public class Point
{
    public int x, y;

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

```

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

```

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

```

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

Members

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

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

MEMBER	DESCRIPTION
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the following table.

ACCESSIBILITY	MEANING
<code>public</code>	Access not limited
<code>protected</code>	Access limited to this class or classes derived from this class
<code>internal</code>	Access limited to this program
<code>protected internal</code>	Access limited to this program or classes derived from this class
<code>private</code>	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 then be used in the body of the class

declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond`:

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

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

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

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

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

Base classes

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

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

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

public class Point3D: Point
{
    public int z;

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

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

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

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

Fields

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

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

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

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

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

    private byte r, g, b;

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

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

Methods

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

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

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

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

Parameters

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

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

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

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

```
using System;

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

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

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

```
using System;

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

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

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

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

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

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

is equivalent to writing the following.

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

Method body and local variables

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

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

```
using System;

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

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

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

Static and instance methods

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

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

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

```

class Entity
{
    static int nextSerialNo;
    int serialNo;

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

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

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

```

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

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

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

```

using System;

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

```

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

Virtual, override, and abstract methods

When an instance method declaration includes a `virtual` modifier, the method is said to be a *virtual method*.

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

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

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

inherited virtual method by providing a new implementation of that method.

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

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

```

using System;
using System.Collections;

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

public class Constant: Expression
{
    double value;

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

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

public class VariableReference: Expression
{
    string name;

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

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

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

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

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

```

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

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

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

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

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

```
using System;
using System.Collections;

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

Method overloading

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

```

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

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

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

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

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

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

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

```

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

Other function members

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

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

```

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

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

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

    // Properties...

```

```

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

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

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

// Event...
public event EventHandler Changed;

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

```

Constructors

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

actions required to initialize a class itself when it is first loaded.

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

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

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

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

Properties

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

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

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

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

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

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

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

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

Indexers

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

The `List` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `List` instances with `int` values. For example

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

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

Events

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

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

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

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

```
using System;

class Test
{
    static int changeCount;

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

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

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

Operators

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

All operators must be declared as `public` and `static`.

The `List<T>` class declares two operators, `operator==` and `operator!=`, and thus gives new meaning to expressions that apply those operators to `List` instances. Specifically, the operators define equality of two `List<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `List<int>` instances.

```
using System;

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

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

Destructors

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

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

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

Structs

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

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

```

class Point
{
    public int x, y;

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

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

```

An alternative is to make `Point` a struct.

```

struct Point
{
    public int x, y;

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

```

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

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

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

```

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

```

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

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

Arrays

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

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

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

```
using System;

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

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

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

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

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

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

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

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

initializes an `int[]` with three elements.

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

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

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

Both of the previous examples are equivalent to the following:

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

Interfaces

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

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

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

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

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

interface IComboBox: ITextBox, IListBox {}
```

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

```

interface IDataBound
{
    void Bind(Binder b);
}

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

```

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

Enums

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

```

using System;

enum Color
{
    Red,
    Green,
    Blue
}

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

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

```

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

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

```

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

```

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

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

```

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

```

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

```
Color c = 0;
```

Delegates

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

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

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

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

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

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

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

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

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

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

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

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

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

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

Attributes

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

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

```
using System;

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

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

    public string Url {
        get { return url; }
    }

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

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

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

This example attaches a `HelpAttribute` to the `Widget` class and another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

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

```
using System;
using System.Reflection;

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

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

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

Pattern Matching for C# 7

3/13/2021 • 15 minutes to read • [Edit Online](#)

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. The basic features are: [record types](#), which are types whose semantic meaning is described by the shape of the data; and pattern matching, which is a new expression form that enables extremely concise multilevel decomposition of these data types. Elements of this approach are inspired by related features in the programming languages [F#](#) and [Scala](#).

Is expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
  : relational_expression 'is' pattern
  ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Patterns

Patterns are used in the `is` operator and in a *switch_statement* to express the shape of data against which incoming data is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```

pattern
  : declaration_pattern
  | constant_pattern
  | var_pattern
  ;

declaration_pattern
  : type simple_designation
  ;

constant_pattern
  : shift_expression
  ;

var_pattern
  : 'var' simple_designation
  ;

```

Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Declaration pattern

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. If the *simple_designation* is an identifier, it introduces a local variable of the given type named by the given identifier. That local variable is *definitely assigned* when the result of the pattern-matching operation is true.

```

declaration_pattern
  : type simple_designation
  ;

```

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype), the result of the `is operator` is `true`. It declares a new local variable named by the *identifier* that is assigned the value of the left-hand operand when the result is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

Note: In C# 7.1 we extend this to permit a pattern-matching operation if either the input type or the type `T` is an open type. This paragraph is replaced by the following:

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, or if either `E` or `T` is an open type. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

The declaration pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v }
```

With the slightly more concise

```
if (expr is Type v) { // code using v }
```

It is an error if *type* is a nullable value type.

The declaration pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v }
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block.

Constant pattern

```
constant_pattern
: shift_expression
;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant, or a `typeof` expression.

If both *e* and *c* are of integral types, the pattern is considered matched if the result of the expression `e == c` is `true`.

Otherwise the pattern is considered matching if `object.Equals(e, c)` returns `true`. In this case it is a compile-time error if the static type of *e* is not *pattern compatible* with the type of the constant.

Var pattern

```
var_pattern
: 'var' simple_designation
;
```

An expression *e* matches a *var_pattern* always. In other words, a match to a *var pattern* always succeeds. If the *simple_designation* is an identifier, then at runtime the value of *e* is bound to a newly introduced local variable. The type of the local variable is the static type of *e*.

It is an error if the name `var` binds to a type.

Switch statement

The `switch` statement is extended to select for execution the first block having an associated pattern that matches the *switch expression*.

```

switch_label
: 'case' complex_pattern case_guard? ':'
| 'case' constant_expression case_guard? ':'
| 'default' ':'
;

case_guard
: 'when' expression
;

```

The order in which patterns are matched is not defined. A compiler is permitted to match patterns out of order, and to reuse the results of already matched patterns to compute the result of matching of other patterns.

If a *case-guard* is present, its expression is of type `bool`. It is evaluated as an additional condition that must be satisfied for the case to be considered satisfied.

It is an error if a *switch_label* can have no effect at runtime because its pattern is subsumed by previous cases.
[TODO: We should be more precise about the techniques the compiler is required to use to reach this judgment.]

A pattern variable declared in a *switch_label* is definitely assigned in its case block if and only if that case block contains precisely one *switch_label*.

[TODO: We should specify when a *switch block* is reachable.]

Scope of pattern variables

The scope of a variable declared in a pattern is as follows:

- If the pattern is a case label, then the scope of the variable is the *case block*.

Otherwise the variable is declared in an *is_pattern* expression, and its scope is based on the construct immediately enclosing the expression containing the *is_pattern* expression as follows:

- If the expression is in an expression-bodied lambda, its scope is the body of the lambda.
- If the expression is in an expression-bodied method or property, its scope is the body of the method or property.
- If the expression is in a `when` clause of a `catch` clause, its scope is that `catch` clause.
- If the expression is in an *iteration_statement*, its scope is just that statement.
- Otherwise if the expression is in some other statement form, its scope is the scope containing the statement.

For the purpose of determining the scope, an *embedded_statement* is considered to be in its own scope. For example, the grammar for an *if_statement* is

```

if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
;

```

So if the controlled statement of an *if_statement* declares a pattern variable, its scope is restricted to that *embedded_statement*.

```
if (x) M(y is var z);
```

In this case the scope of `z` is the embedded statement `M(y is var z);`.

Other cases are errors for other reasons (e.g. in a parameter's default value or an attribute, both of which are an error because those contexts require a constant expression).

In C# 7.3 we added the following contexts in which a pattern variable may be declared:

- If the expression is in a *constructor initializer*, its scope is the *constructor initializer* and the constructor's body.
- If the expression is in a field initializer, its scope is the *equals_value_clause* in which it appears.
- If the expression is in a query clause that is specified to be translated into the body of a lambda, its scope is just that expression.

Changes to syntactic disambiguation

There are situations involving generics where the C# grammar is ambiguous, and the language spec says how to resolve those ambiguities:

7.6.5.2 Grammar ambiguities

The productions for *simple-name* (§7.6.3) and *member-access* (§7.6.5) can give rise to ambiguities in the grammar for expressions. For example, the statement:

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

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

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or `> pointer-member-access`, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§3.8). The statement

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

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

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

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

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

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

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace-or-type-name* with a *type-argument-list*.

There are a number of changes being introduced in C# 7 that make these disambiguation rules no longer sufficient to handle the complexity of the language.

Out variable declarations

It is now possible to declare a variable in an out argument:

```
M(out Type name);
```

However, the type may be generic:

```
M(out A<B> name);
```

Since the language grammar for the argument uses *expression*, this context is subject to the disambiguation rule. In this case the closing `>` is followed by an *identifier*, which is not one of the tokens that permits it to be treated as a *type-argument-list*. I therefore propose to add *identifier* to the set of tokens that triggers the disambiguation to a *type-argument-list*.

Tuples and deconstruction declarations

A tuple literal runs into exactly the same issue. Consider the tuple expression

```
(A < B, C > D, E < F, G > H)
```

Under the old C# 6 rules for parsing an argument list, this would parse as a tuple with four elements, starting with `A < B` as the first. However, when this appears on the left of a deconstruction, we want the disambiguation triggered by the *identifier* token as described above:

```
(A<B,C> D, E<F,G> H) = e;
```

This is a deconstruction declaration which declares two variables, the first of which is of type `A<B,C>` and named `D`. In other words, the tuple literal contains two expressions, each of which is a declaration expression.

For simplicity of the specification and compiler, I propose that this tuple literal be parsed as a two-element tuple wherever it appears (whether or not it appears on the left-hand-side of an assignment). That would be a natural result of the disambiguation described in the previous section.

Pattern-matching

Pattern matching introduces a new context where the expression-type ambiguity arises. Previously the right-hand-side of an `is` operator was a type. Now it can be a type or expression, and if it is a type it may be followed by an identifier. This can, technically, change the meaning of existing code:

```
var x = e is T < A > B;
```

This could be parsed under C#6 rules as

```
var x = ((e is T) < A) > B;
```

but under C#7 rules (with the disambiguation proposed above) would be parsed as

```
var x = e is T<A> B;
```

which declares a variable `B` of type `T<A>`. Fortunately, the native and Roslyn compilers have a bug whereby they give a syntax error on the C#6 code. Therefore this particular breaking change is not a concern.

Pattern-matching introduces additional tokens that should drive the ambiguity resolution toward selecting a type. The following examples of existing valid C#6 code would be broken without additional disambiguation rules:

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;            // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];             // [
```

Proposed change to the disambiguation rule

I propose to revise the specification to change the list of disambiguating tokens from

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || &
```

And, in certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case`, or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.

Modified disambiguation rule

The revised disambiguation rule would be something like this

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined, to see if it is

- One of `()] } : ; , . ? == != | ^ && || & [`; or
- One of the relational operators `< > <= >= is as`; or
- A contextual query keyword appearing inside a query expression; or
- In certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case` or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.

If the following token is among this list, or an identifier in such a context, then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a

namespace-or-type-name (§3.8).

Breaking changes due to this proposal

No breaking changes are known due to this proposed disambiguation rule.

Interesting examples

Here are some interesting results of these disambiguation rules:

The expression `(A < B, C > D)` is a tuple with two elements, each a comparison.

The expression `(A<B,C> D, E)` is a tuple with two elements, the first of which is a declaration expression.

The invocation `M(A < B, C > D, E)` has three arguments.

The invocation `M(out A<B,C> D, E)` has two arguments, the first of which is an `out` declaration.

The expression `e is A C` uses a declaration expression.

The case label `case A C:` uses a declaration expression.

Some examples of pattern matching

Is-As

We can replace the idiom

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

With the slightly more concise and direct

```
if (expr is Type v) {
    // code using v
}
```

Testing nullable

We can replace the idiom

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

With the slightly more concise and direct

```
if (x?.y?.z is Type value) {
    // code using value
}
```

Arithmetic simplification

Suppose we define a set of recursive types to represent expressions (per a separate proposal):

```

abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;

```

Now we can define a function to compute the (unreduced) derivative of an expression:

```

Expr Deriv(Expr e)
{
    switch (e) {
        case X(): return Const(1);
        case Const(*): return Const(0);
        case Add(var Left, var Right):
            return Add(Deriv(Left), Deriv(Right));
        case Mult(var Left, var Right):
            return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
        case Neg(var Value):
            return Neg(Deriv(Value));
    }
}

```

An expression simplifier demonstrates positional patterns:

```

Expr Simplify(Expr e)
{
    switch (e) {
        case Mult(Const(0), *): return Const(0);
        case Mult(*, Const(0)): return Const(0);
        case Mult(Const(1), var x): return Simplify(x);
        case Mult(var x, Const(1)): return Simplify(x);
        case Mult(Const(var l), Const(var r)): return Const(l*r);
        case Add(Const(0), var x): return Simplify(x);
        case Add(var x, Const(0)): return Simplify(x);
        case Add(Const(var l), Const(var r)): return Const(l+r);
        case Neg(Const(var k)): return Const(-k);
        default: return e;
    }
}

```

C# walkthroughs

11/2/2020 • 2 minutes to read • [Edit Online](#)

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

This section contains links to C# programming walkthroughs.

In this section

- [Process asynchronous tasks as they complete](#)
Shows how to create an asynchronous solution by using `async` and `await`.
- [Creating a Windows Runtime Component in C# or Visual Basic and Calling it from JavaScript](#)
Shows how to create a Windows Runtime type, package it in a Windows Runtime component, and then call the component from a Windows 8.x Store app that's built for Windows by using JavaScript.
- [Office Programming \(C# and Visual Basic\)](#)
Shows how to create an Excel workbook and a Word document by using C# and Visual Basic.
- [Creating and Using Dynamic Objects \(C# and Visual Basic\)](#)
Shows how to create a custom object that dynamically exposes the contents of a text file, and how to create a project that uses the `IronPython` library.
- [Authoring a Composite Control with Visual C#](#)
Demonstrates creating a simple composite control and extending its functionality through inheritance.
- [Creating a Windows Forms Control that Takes Advantage of Visual Studio Design-Time Features](#)
Illustrates how to create a custom designer for a custom control.
- [Inheriting from a Windows Forms Control with Visual C#](#)
Demonstrates creating a simple inherited button control. This button inherits functionality from the standard Windows Forms button and exposes a custom member.
- [Debugging Custom Windows Forms Controls at Design Time](#)
Describes how to debug the design-time behavior of your custom control.
- [Walkthrough: Perform common tasks using designer actions](#)
Demonstrates some of the commonly performed tasks such as adding or removing a tab on a `TabControl`, docking a control to its parent, and changing the orientation of a `SplitContainer` control.
- [Writing Queries in C# \(LINQ\)](#)
Demonstrates the C# language features that are used to write LINQ query expressions.
- [Manipulating Data \(C#\) \(LINQ to SQL\)](#)
Describes a LINQ to SQL scenario for adding, modifying, and deleting data in a database.
- [Simple Object Model and Query \(C#\) \(LINQ to SQL\)](#)
Demonstrates how to create an entity class and a simple query to filter the entity class.
- [Using Only Stored Procedures \(C#\) \(LINQ to SQL\)](#)
Demonstrates how to use LINQ to SQL to access data by executing only stored procedures.
- [Querying Across Relationships \(C#\) \(LINQ to SQL\)](#)
Demonstrates the use of LINQ to SQL associations to represent foreign-key relationships in a database.

- [Writing a Visualizer in C#](#)

Shows how to write a simple visualizer by using C#.

Related sections

- [Deployment Samples and Walkthroughs](#)

Provides step-by-step examples of common deployment scenarios.

See also

- [C# Programming Guide](#)
- [Visual Studio Samples](#)