

Partial Classes and Methods (C# Programming Guide)

Article • 01/25/2022 • 6 minutes to read • [20 contributors](#)



In this article

[Partial Classes](#)

[Examples](#)

[Partial Methods](#)

[C# Language Specification](#)

[See also](#)


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.
- When using [source generators](#) to generate additional functionality in a class.

To split a class definition, use the [partial](#) keyword modifier, as shown here:

C#	 Copy
<pre>public partial class Employee { public void DoWork() { } }</pre>	

```
}

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.


C#

 Copy


```
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:

C#	 Copy
<pre>[SerializableAttribute] partial class Moon { }</pre> <pre>[ObsoleteAttribute] partial class Moon { }</pre>	

They are equivalent to the following declarations:

C#	 Copy
<pre>[SerializableAttribute] [ObsoleteAttribute] class Moon { }</pre>	

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:

C#	 Copy
<pre>partial class Earth : Planet, IRotate { }</pre> <pre>partial class Earth : IRevolve { }</pre>	


They are equivalent to the following declarations:

C#	 Copy
<pre>class Earth : Planet, IRotate, IRevolve { }</pre>	

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:

C#	 Copy
<pre>public partial class A { } //public class A { } // Error, must also be marked partial</pre>	

- 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:


C#	 Copy
<pre>partial class ClassWithNestedClass { partial class NestedClass { } } partial class ClassWithNestedClass { partial class NestedClass { } }</pre>	

- 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](#).

Examples

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.

C#	 Copy
<pre>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</pre>	

The following example shows that you can also develop partial structs and interfaces.

C#	 Copy
<pre>partial interface ITest { void Interface_Test(); }</pre>	

```
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. 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. That implementation may be supplied by a *source generator*.

Partial methods enable the implementer of one part of a class to declare a method. The implementer of another part of the class can define that method. There are two scenarios where this is useful: templates that generate boilerplate code, and source generators.

- **Template code:** The template reserves a method name and signature so that generated code can call the method. These methods follow the restrictions that enable a developer to decide whether to implement the method. 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.

- **Source generators:** Source generators provide an implementation for methods. The human developer can add the method declaration (often with attributes read by the source generator). The developer can write code that calls these methods. The source generator runs during compilation and provides the implementation. In this scenario, the restrictions for partial methods that may not be implemented often aren't followed.

C#	 Copy
<pre>// Definition in file1.cs partial void OnNameChanged(); // Implementation in file2.cs partial void OnNameChanged() { // method body }</pre>	

- 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.

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 \(Type\)](#)

Recommended content

[Access Modifiers - C# Programming Guide](#)

All types and type members in C# have an accessibility level which controls whether they can be used from other code. Review this list of access modifiers.

[Static Classes and Static Class Members - C# Programming Guide](#)

Static classes cannot be instantiated in C#. You access the members of a static class by using the class name itself.

[Extension Methods - C# Programming Guide](#)

Extension methods in C# enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

[Lambda expressions - C# reference](#)

Learn about C# lambda expressions that are used to create anonymous functions.

[=> operator - C# reference](#)

Learn about the C# => operator that is used in lambda expressions and expression body definitions.

[override modifier - C# Reference](#)

[override modifier - C# Reference](#)

[base keyword - C# Reference](#)

Learn about the base keyword, which is used to access members of the base class from within a derived class in C#.

[Using Properties - C# Programming Guide](#)

These examples illustrate using properties in C#. See how the get and set accessors implement read and write access and find out about uses for properties.

Show more 