Visual Studio 2010 - Visual C#

**Partial Classes and Methods (C# Programming Guide)**

It is possible to split the definition of a [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or a [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx), an [interface](http://msdn.microsoft.com/en-us/library/87d83y5b.aspx) 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.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifPartial 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](http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx) keyword modifier, as shown here:

C#

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.

|  |
| --- |
| **Description: NoteNote** |
| 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#

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#

[SerializableAttribute]

partial class Moon { }

[ObsoleteAttribute]

partial class Moon { }

They are equivalent to the following declarations:

C#

[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:

C#

partial class Earth : Planet, IRotate { }

partial class Earth : IRevolve { }

They are equivalent to the following declarations:

C#

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:

C#

public partial class A { }

//public class tcA { } // 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:

C#

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](http://msdn.microsoft.com/en-us/library/yzh058ae.aspx)
  + [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx)
  + [protected](http://msdn.microsoft.com/en-us/library/bcd5672a.aspx)
  + [internal](http://msdn.microsoft.com/en-us/library/7c5ka91b.aspx)
  + [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5.aspx)
  + [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw.aspx)
  + base class
  + [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx) modifier (nested parts)
  + generic constraints

For more information, see [Constraints on Type Parameters (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/d5x73970.aspx).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifExample 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**

C#

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

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifExample 2

**Description**

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

**Code**

C#

partial interface ITest

{

void Interface\_Test();

}

partial interface ITest

{

void Interface\_Test2();

}

partial struct S1

{

void Struct\_Test() { }

}

partial struct S1

{

void Struct\_Test2() { }

}

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifPartial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may 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.

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](http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx) and the method must return [void](http://msdn.microsoft.com/en-us/library/yah0tteb.aspx).
* Partial methods can have [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) but not [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameters.
* Partial methods are implicitly [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx), and therefore they cannot be [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4.aspx).
* Partial methods cannot be [extern](http://msdn.microsoft.com/en-us/library/e59b22c5.aspx), because the presence of the body determines whether they are defining or implementing.
* Partial methods can have [static](http://msdn.microsoft.com/en-us/library/98f28cdx.aspx) and [unsafe](http://msdn.microsoft.com/en-us/library/chfa2zb8.aspx) 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](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) to a partial method that has been defined and implemented, but not to a partial method that has only been defined.