Visual Studio 2010 - Visual C#

**Extension Methods (C# Programming Guide)**

Updated: October 2010

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 a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually 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](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx) and [System.Collections.Generic.IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) types. To use the standard query operators, first bring them into scope with a using System.Linq directive. Then any type that implements [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) appears to have instance methods such as [GroupBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.groupby.aspx), [OrderBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.orderby.aspx), [Average](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.average.aspx), and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) type such as [List<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx) or [Array](http://msdn.microsoft.com/en-us/library/system.array.aspx).

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 is not a requirement for extension methods. For more information, see [Lambda Expressions (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397687.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl21_code');" \o "Copy Code)

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, and the parameter is preceded by the [this](http://msdn.microsoft.com/en-us/library/dk1507sz.aspx) 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](http://msdn.microsoft.com/en-us/library/system.string.aspx) class. Note that it is defined inside a non-nested, non-generic static class:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl25_code');" \o "Copy Code)

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:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl26_code');" \o "Copy Code)

using ExtensionMethods;

And it can be called from an application by using this syntax:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl27_code');" \o "Copy Code)

string s = "Hello Extension Methods";

int i = s.WordCount();

In your code you invoke the extension method with instance method syntax. However, the intermediate language (IL) generated by the compiler translates your code into a call on the static method. Therefore, the principle of encapsulation is not really being violated. In fact, 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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb311042.aspx).

In general, you will 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:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_code');" \o "Copy Code)

using System.Linq;

(You may also have to add a reference to System.Core.dll.) You will notice that the standard query operators now appear in IntelliSense as additional methods available for most [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) types.

|  |
| --- |
| **Description: NoteNote** |
| Although standard query operators do not appear in IntelliSense for [String](http://msdn.microsoft.com/en-us/library/system.string.aspx), they are still available. |

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

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

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 cannot find an instance method with a matching signature, it will bind to a matching extension method if one exists.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl70_ctl00_ctl00_code');" \o "Copy Code)

// 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(object, int)

a.MethodA("hello"); // Extension.MethodA(object, 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

// nethod 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(object, 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()

\*/

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifGeneral Guidelines

In general, we recommend that you implement extension methods sparingly and only when you have to. Whenever possible, client code that must extend an existing type should do so by creating a new type derived from the existing type. For more information, see [Inheritance (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173149.aspx).

When using an extension method to extend a type whose source code you cannot change, 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 two 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 will all be brought into scope by the using Extensions; directive.

**Extension Methods**

***Extension methods*** are static methods that can be invoked using instance method syntax. In effect, extension methods make it possible to extend existing types and constructed types with additional methods.

**Note** Extension methods are less discoverable and more limited in functionality than instance methods. For those reasons, it is recommended that extension methods be used sparingly and only in situations where instance methods are not feasible or possible. Extension members of other kinds, such as properties, events, and operators, are being considered but are currently not supported.

**Declaring Extension Methods**

Extension methods are declared by specifying the keyword **this** as a modifier on the first parameter of the methods. Extension methods can only be declared in non-generic, non-nested static classes. The following is an example of a static class that declares two extension methods:

using System;

using System.Collections.Generic;

using Acme.Utilities;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

string s = "12345";

int i = s.ToInt32();

Console.WriteLine("i = " + i);

int[] d = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int[] a = d.Slice(4, 3);

for (int index = 0; index < a.Length; index++)

{

Console.WriteLine("a[" + index + "] = " + a[index]);

}

}

}

}

namespace Acme.Utilities

{

static class Extensions

{

internal static int ToInt32(this string s)

{

return Int32.Parse(s);

}

internal static T[] Slice<T>(this T[] source, int index, int count)

{

if(index < 0 || count < 0 || ((source.Length - index) < count))

{

throw new ArgumentException();

}

T[] result = new T[count];

Array.Copy(source, index, result, 0, count);

return result;

}

}

}

/\*

Output:

i = 12345

a[0] = 5

a[1] = 6

a[2] = 7

\*/