Visual Studio 2010 - Visual C#

**Named and Optional Arguments (C# Programming Guide)**

Visual C# 2010 introduces named and optional arguments. *Named arguments* enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than with the parameter's 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, when used together, enable you to supply arguments for only a few parameters from a list of optional parameters. This capability greatly facilitates calls to COM interfaces such as the Microsoft Office Automation APIs.

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

Named arguments free you from the need to remember or to look up 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 calculates body mass index (BMI) can be called in the standard way by sending arguments for weight and height by position, in the order defined by the function.

CalculateBMI(123, 64);

If you do not remember the order of the parameters but you do know their names, you can send the arguments in either order, weight first or height first.

CalculateBMI(weight: 123, height: 64);

CalculateBMI(height: 64, weight: 123);

Named arguments also improve the readability of your code by identifying what each argument represents.

A named argument can follow positional arguments, as shown here.

CalculateBMI(123, height: 64);

However, a positional argument cannot follow a named argument. The following statement causes a compiler error.

//CalculateBMI(weight: 123, 64);

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

The following code implements the examples from this section.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl31_ctl00_ctl00_code');" \o "Copy Code)

class NamedExample

{

static void Main(string[] args)

{

// The method can be called in the normal way, by using positional arguments.

Console.WriteLine(CalculateBMI(123, 64));

// Named arguments can be supplied for the parameters in either order.

Console.WriteLine(CalculateBMI(weight: 123, height: 64));

Console.WriteLine(CalculateBMI(height: 64, weight: 123));

// Positional arguments cannot follow named arguments.

// The following statement causes a compiler error.

//Console.WriteLine(CalculateBMI(weight: 123, 64));

// Named arguments can follow positional arguments.

Console.WriteLine(CalculateBMI(123, height: 64));

}

static int CalculateBMI(int weight, int height)

{

return (weight \* 703) / (height \* height);

}

}

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

The definition of a method, constructor, indexer, or delegate can specify that its parameters are required or that they are 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](http://msdn.microsoft.com/en-us/library/sbbt4032.aspx) or a [struct](http://msdn.microsoft.com/en-us/library/saxz13w4.aspx);
* an expression of the form [default(ValType)](http://msdn.microsoft.com/en-us/library/xwth0h0d.aspx), 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 are not supported. For example, in the following code, instance method ExampleMethod is defined with one required and two optional parameters.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl32_ctl00_ctl03_code');" \o "Copy Code)

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.

**Optional parameters in ExampleMethod**



|  |
| --- |
| **Description: NoteNote** |
| You can also declare optional parameters by using the .NET [OptionalAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.optionalattribute.aspx) class. **OptionalAttribute** parameters do not require a default value. |

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

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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl00_code');" \o "Copy Code)

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("{0}: {1}, {2}, and {3}.", \_name, required, optionalstr,

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.

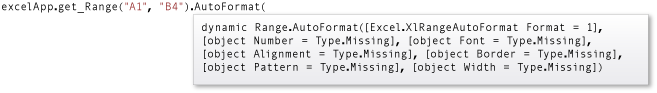
}

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

Named and optional arguments, along with support for dynamic objects and other enhancements, greatly improve interoperability with COM APIs, such as Office Automation APIs.

For example, the [AutoFormat](http://go.microsoft.com/fwlink/?LinkId=148201) method in the Microsoft Office Excel [Range](http://go.microsoft.com/fwlink/?LinkId=148196) interface has seven parameters, all of which are optional. These parameters are shown in the following illustration.

**AutoFormat parameters**



In C# 3.0 and earlier versions, an argument is required for each parameter, as shown in the following example.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_ctl00_ctl03_code');" \o "Copy Code)

// 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, 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 do not want to change the parameter's default value. In the following call, a value is specified for only one of the seven parameters.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_ctl00_ctl04_code');" \o "Copy Code)

// 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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/dd264738.aspx) and [How to: Access Office Interop Objects by Using Visual C# 2010 Features (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/dd264733.aspx).

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

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifC# Language Specification

For more information, see the [C# Language Specification](http://msdn.microsoft.com/en-us/library/ms228593.aspx). The language specification is the definitive source for C# syntax and usage.