Visual Studio 2010 - Visual C#

**Using Type dynamic (C# Programming Guide)**

Visual C# 2010 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.

C#

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();

}

C#

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**.

C#

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, and constructor calls that include arguments of type **dynamic**. For example, the type of testInstance in the following declaration is ExampleClass, not **dynamic**.

C#

var testInstance = new ExampleClass(d);

Conversion examples are shown in the following section, "Conversions."

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

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.

C#

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**.

C#

int i = d1;

string str = d2;

DateTime dt = d3;

System.Diagnostics.Process[] procs = d4;

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

C#

// 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);

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

Dynamic Language Runtime

The dynamic language runtime (DLR) is a new API 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](http://msdn.microsoft.com/en-us/library/dd233052.aspx).

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

COM Interop

Visual C# 2010 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](http://msdn.microsoft.com/en-us/library/dd264739.aspx).

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 [/link (C# Compiler Options)](http://msdn.microsoft.com/en-us/library/dd264728.aspx) 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.

C#

// Before the introduction of dynamic.

((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";

Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];

C#

// 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];

Visual Studio 2010 - Visual C#

**dynamic (C# Reference)**

The **dynamic** type enables the operations in which it occurs to 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, and also to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type **dynamic** behaves like type **object** in most circumstances. However, 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. IntelliSense shows **dynamic** for dyn and **object** for obj.

C#

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.

C#

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**.

Context

The **dynamic** keyword can appear directly or as a component of a constructed type in the following situations:

* In declarations, as the type of a property, field, indexer, parameter, return value, local variable, or type constraint. The following class definition uses **dynamic** in several different declarations.

C#

class ExampleClass

{

// A dynamic field.

static dynamic field;

// A dynamic property.

dynamic prop { get; set; }

// A dynamic return type and a dynamic paramater type.

public dynamic exampleMethod(dynamic d)

{

// A dynamic local variable.

dynamic local = "Local variable";

int two = 2;

if (d is int)

{

return local;

}

else

{

return two;

}

}

}

* In explicit type conversions, as the target type of a conversion.

C#

static void convertToDynamic()

{

dynamic d;

int i = 20;

d = (dynamic)i;

Console.WriteLine(d);

string s = "Example string.";

d = (dynamic)s;

Console.WriteLine(d);

DateTime dt = DateTime.Today;

d = (dynamic)dt;

Console.WriteLine(d);

}

// Results:

// 20

// Example string.

// 2/17/2009 9:12:00 AM

* In any context where types serve as values, such as on the right side of an **is** operator or an **as** operator, or as the argument to **typeof** as part of a constructed type. For example, **dynamic** can be used in the following expressions.

C#

int i = 8;

dynamic d;

// With the is operator.

// The dynamic type behaves like object. The following

// expression returns true unless someVar has the value null.

if (someVar is dynamic) { }

// With the as operator.

d = i as dynamic;

// With typeof, as part of a constructed type.

Console.WriteLine(typeof(List<dynamic>));

// The following statement causes a compiler error.

//Console.WriteLine(typeof(dynamic));

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

Example

The following example uses **dynamic** in several declarations. The Main method also contrasts compile-time type checking with run-time type checking.

C#

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, itdoes 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

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#

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#

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#

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.

}

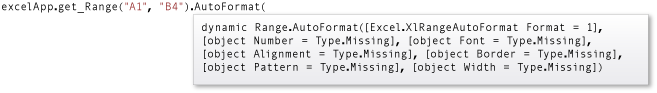
Description: http://i.msdn.microsoft.com/Global/Images/clear.gif

COM 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#

// 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#

// 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.

Visual Studio 2010 - Visual C#

**How to: Use Indexed Properties in COM Interop Programming (C# Programming Guide)**

*Indexed properties* improve the way in which COM properties that have parameters are consumed in C# programming. Indexed properties work together with other features introduced in Visual C# 2010, such as [named and optional arguments](http://msdn.microsoft.com/en-us/library/dd264739.aspx), a new type ([dynamic](http://msdn.microsoft.com/en-us/library/dd264741.aspx)), and [embedded type information](http://msdn.microsoft.com/en-us/library/dd409610.aspx), 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](http://go.microsoft.com/fwlink/?LinkId=166053) 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.

C#

// 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:

C#

// Visual C# 2010.

var excelApp = new Excel.Application();

// . . .

Excel.Range targetRange = excelApp.Range["A1"];

|  |
| --- |
| **Description: NoteNote** |
| The previous example also uses the [optional arguments](http://msdn.microsoft.com/en-us/library/dd264739.aspx) feature, introduced in Visual C# 2010, which enables you to omit **Type.Missing**. |

Similarly, to set the value of the **Value** property of a [Range](http://go.microsoft.com/fwlink/?LinkId=179211) object in Visual C# 2008 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. Before Visual C# 2010, C# allowed only one argument. Therefore, instead of using a regular set method, you had to either use the **set\_Value** method or a different property, [Value2](http://go.microsoft.com/fwlink/?LinkId=166050). The following examples illustrate these techniques. Both set the value of the A1 cell to Name.

C#

// Visual C# 2008.

targetRange.set\_Value(Type.Missing, "Name");

// Or

targetRange.Value2 = "Name";

Indexed properties enable you to write the following code instead.

C#

// 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 Visual C# 2010 Features (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/dd264733.aspx).

C#

// 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";

}

}

}

Visual Studio 2010 - Visual C#

**/link (C# Compiler Options)**

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

/link:fileList

// -or-

/l:fileList

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

*fileList*

Required. Comma-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks.

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

The **/link** 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 (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/dd409610.aspx).

Using the **/link** option is especially useful when you are 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 **/link** 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 **/link** option embeds only interfaces, structures, and delegates. Embedding COM classes is not supported.

|  |
| --- |
| **Description: NoteNote** |
| 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. |

To set the **/link** option in Visual Studio, add an assembly reference and set the **Embed Interop Types** property to **true**. The default for the **Embed Interop Types** property is **false**.

If you link to a COM assembly (Assembly A) which itself references another COM assembly (Assembly B), you also have to link to Assembly B if either of the following is true:

* A type from Assembly A inherits from a type or implements an interface from Assembly B.
* A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Like the [/reference](http://msdn.microsoft.com/en-us/library/yabyz3h4.aspx) compiler option, the **/link** compiler option uses the Csc.rsp response file, which references frequently used .NET Framework assemblies. Use the [/noconfig](http://msdn.microsoft.com/en-us/library/8hww4s6c.aspx) compiler option if you do not want the compiler to use the Csc.rsp file.

The short form of **/link** is **/l**.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifGenerics and Embedded Types

The following sections describe the limitations on using generic types in applications that embed interop types.

**Generic Interfaces**

Generic interfaces that are embedded from an interop assembly cannot be used. This is shown in the following example.

C#

// The following code causes an error if ISampleInterface is an embedded interop type.

ISampleInterface<SampleType> sample;

**Types That Have Generic Parameters**

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 does not apply to interfaces. For example, consider the [Range](http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.range.aspx) interface that is defined in the [Microsoft.Office.Interop.Excel](http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.aspx) assembly. If a library embeds interop types from the [Microsoft.Office.Interop.Excel](http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.aspx) assembly and exposes a method that returns a generic type that has a parameter whose type is the [Range](http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.range.aspx) interface, that method must return a generic interface, as shown in the following code example.

C#

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() {

...

}

// The following code is valid for calls from a client assembly.

public IList<Range> GetRange2() {

...

}

}

In the following example, client code can call the method that returns the [IList](http://msdn.microsoft.com/en-us/library/system.collections.ilist.aspx) generic interface without error.

C#

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();

}

}

Example

The following code compiles source file OfficeApp.cs and reference assemblies from COMData1.dll and COMData2.dll to produce OfficeApp.exe.

C#

csc /link:COMData1.dll,COMData2.dll /out:OfficeApp.exe OfficeApp.cs

Visual Studio 2010 - Visual C#

**/langversion (C# Compiler Options)**

Causes the compiler to accept only syntax that is included in the chosen C# language specification.

/langversion:option

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

option

The following values are valid:

|  |  |
| --- | --- |
| **Option** | **Meaning** |
| default | The compiler accepts all valid language syntax. |
| ISO-1 | The compiler accepts only syntax that is included in the ISO/IEC 23270:2003 C# language specification. |
| ISO-2 | The compiler accepts only syntax that is included in the ISO/IEC 23270:2006 C# language specification. This specification is available on the [ISO](http://go.microsoft.com/fwlink/?LinkId=144406) Web site. |
| 3 | The compiler accepts only syntax that is included in the version 3.0 [C# Language Specification](http://msdn.microsoft.com/en-us/library/ms228593.aspx). |

Visual Studio 2010 - Visual C#

**/appconfig (C# Compiler Options)**

The **/appconfig** 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.

/appconfig:file

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

file

Required. The application configuration file that contains assembly binding settings.

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

One use of **/appconfig** 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 **/appconfig** 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.

|  |
| --- |
| **Description: NoteNote** |
| If you are using the Microsoft Build Engine (MSBuild) to build your application, you can set the **/appconfig** compiler option by adding a property tag to the .csproj file. 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. |

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

Example

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 **/appconfig** 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>