C# 4.0

New C# Features in the .NET Framework 4

[Chris Burrows](http://msdn.microsoft.com/magazine/ee532098.aspx?sdmr=ChrisBurrows&sdmi=authors)

Since its initial release in 2002, the C# programming language has been improved to enable programmers to write clearer, more maintainable code. The enhancements have come from the addition of features such as generic types, nullable value types, lambda expressions, iterator methods, partial classes and a long list of other useful language constructs. And, often, the changes were accompanied by giving the Microsoft .NET Framework libraries corresponding support.

This trend toward increased usability continues in C# 4.0. The additions make common tasks involving generic types, legacy interop and working with dynamic object models much simpler. This article aims to give a high-level survey of these new features. I’ll begin with generic variance and then look at the legacy and dynamic interop features.

**Covariance and Contravariance**

Covariance and contravariance are best introduced with an example, and the best is in the framework. In System.Collections.Generic, IEnumerable<T> and IEnumerator <T> represent, respectively, an object that’s a sequence of T’s and the enumerator (or iterator) that does the work of iterating the sequence. These interfaces have done a lot of heavy lifting for a long time, because they support the implementation of the foreach loop construct. In C# 3.0, they became even more prominent because of their central role in LINQ and LINQ to Objects—they’re the .NET interfaces to represent sequences.

So if you have a class hierarchy with, say, an Employee type and a Manager type that derives from it (managers are employees, after all), then what would you expect the following code to do?

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl02_code');" \o "Copy Code)

IEnumerable<Manager> ms = GetManagers();

IEnumerable<Employee> es = ms;

It seems as though one ought to be able to treat a sequence of Managers as though it were a sequence of Employees. But in C# 3.0, the assignment will fail; the compiler will tell you there’s no conversion. After all, it has no idea what the semantics of IEnumerable<T> are. This could be any interface, so for any arbitrary interface IFoo<T>, why would an IFoo<Manager> be more or less substitutable for an IFoo<Employee>?

In C# 4.0, though, the assignment works because IEnumerable<T>, along with a few other interfaces, has changed, an alteration enabled by new support in C# for covariance of type parameters.

IEnumerable<T> is eligible to be more special than the arbitrary IFoo<T> because, though it’s not obvious at first glance, members that use the type parameter T (GetEnumerator in IEnumerable<T> and the Current property in IEnumerator<T>) actually use T only in the position of a return value. So you only get a Manager out of the sequence, and you never put one in.

In contrast, think of List<T>. Making a List<Manager> substitutable for a List<Employee> would be a disaster, because of the following:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl03_code');" \o "Copy Code)

List<Manager> ms = GetManagers();

List<Employee> es = ms; // Suppose this were possible

es.Add(new EmployeeWhoIsNotAManager()); // Uh oh

As this shows, once you think you’re looking at a List<Employee>, you can insert any employee. But the list in question is actually a List<Manager>, so inserting a non-Manager must fail. You’ve lost type safety if you allow this. List<T> cannot be covariant in T.

The new language feature in C# 4.0, then, is the ability to define types, such as the new IEnumerable<T>, that admit conversions among themselves when the type parameters in question bear some relationship to one another. This is what the .NET Framework developers who wrote IEnumerable<T> used, and this is what their code looks like (simplified, of course):

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl04_code');" \o "Copy Code)

public interface IEnumerable<out T> { /\* ... \*/ }

Notice the out keyword modifying the definition of the type parameter, T. When the compiler sees this, it will mark T as covariant and check that, in the definition of the interface, all uses of T are up to snuff (in other words, that they’re used in out positions only—that’s why this keyword was picked).

Why is this called covariance? Well, it’s easiest to see when you start to draw arrows. To be concrete, let’s use the Manager and Employee types. Because there’s an inheritance relationship between these classes, there’s an implicit reference conversion from Manager to Employee:

**Manager → Employee**

And now, because of the annotation of T in IEnumerable<out T>, there’s also an implicit reference conversion from IEnumerable<Manager> to IEnumerable<Employee>. That’s what the annotation provides for:

**IEnumerable<Manager> → IEnumerable<Employee>**

This is called covariance, because the arrows in each of the two examples point in the same direction. We started with two types, Manager and Employee. We made new types out of them, IEnumerable<Manager> and IEnumerable<Employee>. The new types convert the same way as the old ones.

Contravariance is when this happens backward. You might anticipate that this could happen when the type parameter, T, is used only as input, and you’d be right. For example, the System namespace contains an interface called IComparable<T>, which has a single method called CompareTo:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl05_code');" \o "Copy Code)

public interface IComparable<in T> {

bool CompareTo(T other);

}

If you have an IComparable<Employee>, you should be able to treat it as though it were an IComparable<Manager>, because the only thing you can do is put Employees in to the interface. Because a manager is an employee, putting a manager in should work, and it does. The in keyword modifies T in this case, and this scenario functions correctly:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl06_code');" \o "Copy Code)

IComparable<Employee> ec = GetEmployeeComparer();

IComparable<Manager> mc = ec;

This is called contravariance because the arrow got reversed this time:

**Manager → Employee  
IComparable<Manager> ← IComparable<Employee>**

So the language feature here is pretty simple to summarize: You can add the keyword in or out whenever you define a type parameter, and doing so gives you free extra conversions. There are some limitations, though.

First, this works with generic interfaces and delegates only. You can’t declare a generic type parameter on a class or struct in this manner. An easy way to rationalize this is that delegates are very much like interfaces that have just one method, and in any case, classes would often be ineligible for this treatment because of fields. You can think of any field on the generic class as being both an input and an output, depending on whether you write to it or read from it. If those fields involve type parameters, the parameters can be neither covariant nor contravariant.

Second, whenever you have an interface or delegate with a covariant or contravariant type parameter, you’re granted new conversions on that type only when the type arguments, in the usage of the interface (not its definition), are reference types. For instance, because int is a value type, the IEnumerator<int> doesn’t convert to IEnumerator <object>, even though it looks like it should:

**IEnumerator <int>** Description: image: right arrow with slash**IEnumerator <object>**

The reason for this behavior is that the conversion must preserve the type representation. If the int-to-object conversion were allowed, calling the Current property on the result would be impossible, because the value type int has a different representation on the stack than an object reference does. All reference types have the same representation on the stack, however, so only type arguments that are reference types yield these extra conversions.

Very likely, most C# developers will happily use this new language feature—they’ll get more conversions of framework types and fewer compiler errors when using some types from the .NET Framework (IEnumerable<T>, IComparable<T>, Func<T>, Action<T>, among others). And, in fact, anyone designing a library with generic interfaces and delegates is free to use the new in and out type parameters when appropriate to make life easier for their users.

By the way, this feature does require support from the runtime—but the support has always been there. It lay dormant for several releases, however, because no language made use of it. Also, previous versions of C# allowed some limited conversions that were contravariant. Specifically, they let you make delegates out of methods that had compatible return types. In addition, array types have always been covariant. These existing features are distinct from the new ones in C# 4.0, which actually let you define your own types that are covariant and contravariant in some of their type parameters.

**Dynamic Dispatch**

On to the interop features in C# 4.0, starting with what is perhaps the biggest change.

C# now supports dynamic late-binding. The language has always been strongly typed, and it continues to be so in version 4.0. Microsoft believes this makes C# easy to use, fast and suitable for all the work .NET programmers are putting it to. But there are times when you need to communicate with systems not based on .NET.

Traditionally, there were at least two approaches to this. The first was simply to import the foreign model directly into .NET as a proxy. COM Interop provides one example. Since the original release of the .NET Framework, it has used this strategy with a tool called TLBIMP, which creates new .NET proxy types you can use directly from C#.

LINQ-to-SQL, shipped with C# 3.0, contains a tool called SQLMETAL, which imports an existing database into C# proxy classes for use with queries. You’ll also find a tool that imports Windows Management Instrumentation (WMI) classes to C#. Many technologies allow you to write C# (often with attributes) and then perform interop using your handwritten code as basis for external actions, such as LINQ-to-SQL, Windows Communication Foundation (WCF) and serialization.

The second approach abandons the C# type system entirely—you embed strings and data in your code. This is what you do whenever you write code that, say, invokes a method on a JScript object or when you embed a SQL query in your ADO.NET application. You’re even doing this when you defer binding to run time using reflection, even though the interop in that case is with .NET itself.

The dynamic keyword in C# is a response to dealing with the hassles of these other approaches. Let’s start with a simple example—reflection. Normally, using it requires a lot of boilerplate infrastructure code, such as:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07_code');" \o "Copy Code)

object o = GetObject();

Type t = o.GetType();

object result = t.InvokeMember("MyMethod",

BindingFlags.InvokeMethod, null,

o, new object[] { });

int i = Convert.ToInt32(result);

With the dynamic keyword, instead of calling a method MyMethod on some object using reflection in this manner, you can now tell the compiler to please treat o as dynamic and delay all analysis until run time. Code that does that looks like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl08_code');" \o "Copy Code)

dynamic o = GetObject();

int i = o.MyMethod();

It works, and it accomplishes the same thing with code that’s much less convoluted.

The value of this shortened, simplified C# syntax is perhaps more clear if you look at the ScriptObject class that supports operations on a JScript object. The class has an InvokeMember method that has more and different parameters, except in Silverlight, which actually has an Invoke method (notice the difference in the name) with fewer parameters. Neither of these are the same as what you’d need to invoke a method on an IronPython or IronRuby object or on any number of non-C# objects you might come into contact with.

In addition to objects that come from dynamic languages, you’ll find a variety of data models that are inherently dynamic and have different APIs supporting them, such as HTML DOMs, the System.Xml DOM and the XLinq model for XML. COM objects are often dynamic and can benefit from the delay to run time of some compiler analysis.

Essentially, C# 4.0 offers a simplified, consistent view of dynamic operations. To take advantage of it, all you need to do is specify that a given value is dynamic, ensuring that analysis of all operations on the value will be delayed until run time.

In C# 4.0, dynamic is a built-in type, and a special pseudo-keyword signifies it. Note, however, that dynamic is different from var. Variables declared with var actually do have a strong type, but the programmer has left it up to the compiler to figure it out. When the programmer uses dynamic, the compiler doesn’t know what type is being used—the programmer leaves figuring it out up to the runtime.

**Dynamic and the DLR**

The infrastructure that supports these dynamic operations at run time is called the Dynamic Language Runtime (DLR). This new .NET Framework 4 library runs on the CLR, like any other managed library. It’s responsible for brokering each dynamic operation between the language that initiated it and the object it occurs on. If a dynamic operation isn’t handled by the object it occurs on, a runtime component of the C# compiler handles the bind. A simplified and incomplete architecture diagram looks something like **Figure 1**.



Figure 1 **The DLR Runs on Top of the CLR**

The interesting thing about a dynamic operation, such as a dynamic method call, is that the receiver object has an opportunity to inject itself into the binding at run time and can, as a result, completely determine the semantics of any given dynamic operation. For instance, take a look at the following code:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl09_code');" \o "Copy Code)

dynamic d = new MyDynamicObject();

d.Bar("Baz", 3, d);

If MyDynamicObject was defined as shown here, then you can imagine what happens:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl10_code');" \o "Copy Code)

class MyDynamicObject : DynamicObject {

public override bool TryInvokeMember(

InvokeMemberBinder binder,

object[] args, out object result) {

Console.WriteLine("Method: {0}", binder.Name);

foreach (var arg in args) {

Console.WriteLine("Argument: {0}", arg);

}

result = args[0];

return true;

}

}

In fact, the code prints:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl11_code');" \o "Copy Code)

Method: Bar

Argument: Baz

Argument: 3

Argument: MyDynamicObject

By declaring d to be of type dynamic, the code that consumes the MyDynamicObject instance effectively opts out of compile-time checking for the operations d participates in. Use of dynamic means “I don’t know what type this is going to be, so I don’t know what methods or properties there are right now. Compiler, please let them all through and then figure it out when you really have an object at run time.” So the call to Bar compiles even though the compiler doesn’t know what it means. Then at run time, the object itself is asked what to do with this call to Bar. That’s what TryInvokeMember knows how to handle.

Now, suppose that instead of a MyDynamicObject, you used a Python object:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl12_code');" \o "Copy Code)

dynamic d = GetPythonObject();

d.bar("Baz", 3, d);

If the object is the file listed here, then the code also works, and the output is much the same:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl13_code');" \o "Copy Code)

def bar(\*args):

print "Method:", bar.\_\_name\_\_

for x in args:

print "Argument:", x

Under the covers, for each use of a dynamic value, the compiler generates a bunch of code that initializes and uses a DLR CallSite. That CallSite contains all the information needed to bind at run time, including such things as the method name, extra data, such as whether the operation takes place in a checked context, and information about the arguments and their types.

This code, if you had to maintain it, would be every bit as ugly as the reflection code shown earlier or the ScriptObject code or strings that contain XML queries. That’s the point of the dynamic feature in C#—you don’t have to write code like that!

When using the dynamic keyword, your code can look pretty much the way you want: like a simple method invocation, a call to an indexer, an operator, such as +, a cast or even compounds, like += or ++. You can even use dynamic values in statements—for example, if(d) and foreach(var x in d). Short-circuiting is also supported, with code such as d && ShortCircuited or d ?? ShortCircuited.

The value of having the DLR provide a common infrastructure for these sorts of operations is that you’re no longer having to deal with a different API for each dynamic model you’d like to code against—there’s just a single API. And you don’t even have to use it. The C# compiler can use it for you, and that should give you more time to actually write the code you want—the less infrastructure code you have to maintain means more productivity for you.

The C# language provides no shortcuts for defining dynamic objects. Dynamic in C# is all about *consuming and using* dynamic objects. Consider the following:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14_code');" \o "Copy Code)

dynamic list = GetDynamicList();

dynamic index1 = GetIndex1();

dynamic index2 = GetIndex2();

string s = list[++index1, index2 + 10].Foo();

This code compiles, and it contains a lot of dynamic operations. First, there’s the dynamic pre-increment on index1, then the dynamic add with index2. Then a dynamic indexer get is called on list. The product of those operations calls the member Foo. Finally, the total result of the expression is converted to a string and stored in s. That’s five dynamic operations in one line, each dispatched at run time.

The compile-time type of each dynamic operation is itself dynamic, and so the “dynamicness” kind of flows from computation to computation. Even if you hadn’t included dynamic expressions multiple times, there still would be a number of dynamic operations. There are still five in this one line:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl15_code');" \o "Copy Code)

string s = nonDynamicList[++index1, index2 + 10].Foo();

Because the results of the two indexing expressions are dynamic, the index itself is as well. And because the result of the index is dynamic, so is the call to Foo. Then you’re confronted with converting a dynamic value to a string. That happens dynamically, of course, because the object could be a dynamic one that wants to perform some special computation in the face of a conversion request.

Notice in the previous examples that C# allows implicit conversions from any dynamic expression to any type. The conversion to string at the end is implicit and did not require an explicit cast operation. Similarly, any type can be converted to dynamic implicitly.

In this respect, dynamic is a lot like object, and the similarities don’t stop there. When the compiler emits your assembly and needs to emit a dynamic variable, it does so by using the type object and then marking it specially. In some sense, dynamic is kind of an alias for object, but it adds the extra behavior of dynamically resolving operations when you use it.

You can see this if you try to convert between generic types that differ only in dynamic and object; such conversions will always work, because at runtime, an instance of List<dynamic> actually is an instance of List<object>:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl16_code');" \o "Copy Code)

List<dynamic> ld = new List<object>();

You can also see the similarity between dynamic and object if you try to override a method that’s declared with an object parameter:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl17_code');" \o "Copy Code)

class C {

public override bool Equals(dynamic obj) {

/\* ... \*/

}

}

Although it resolves to a decorated object in your assembly, I do like to think of dynamic as a real type, because it serves as a reminder that you can do most things with it that you can do with any other type. You can use it as a type argument or, say, as a return value. For instance, this function definition will let you use the result of the function call dynamically without having to put its return value in a dynamic variable:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl18_code');" \o "Copy Code)

public dynamic GetDynamicThing() {

/\* ... \*/ }

There are a lot more details about the way dynamic is treated and dispatched, but you don’t need to know them to use the feature. The essential idea is that you can write code that looks like C#, and if any part of the code you write is dynamic, the compiler will leave it alone until run time.

I want to cover one final topic concerning dynamic: failure. Because the compiler can’t check whether the dynamic thing you’re using really has the method called Foo, it can’t give you an error. Of course, that doesn’t mean that your call to Foo will work at run time. It may work, but there are a lot of objects that don’t have a method called Foo. When your expression fails to bind at run time, the binder makes its best attempt to give you an exception that’s more or less exactly what the compiler would’ve told you if you hadn’t used dynamic to begin with.

Consider the following code:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl19_code');" \o "Copy Code)

try

{

dynamic d = "this is a string";

d.Foo();

}

catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException e)

{

Console.WriteLine(e.Message);

}

Here I have a string, and strings clearly do not have a method called Foo. When the line that calls Foo executes, the binding will fail and you’ll get a RuntimeBinderException. This is what the previous program prints:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl20_code');" \o "Copy Code)

'string' does not contain a definition for 'Foo'

Which is exactly the error message you, as a C# programmer, expect.

**Named Arguments and Optional Parameters**

In another addition to C#, methods now support optional parameters with default values so that when you call such a method you can omit those parameters. You can see this in action in this Car class:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl21_code');" \o "Copy Code)

class Car {

public void Accelerate(

double speed, int? gear = null,

bool inReverse = false) {

/\* ... \*/

}

}

You can call the method this way:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl22_code');" \o "Copy Code)

Car myCar = new Car();

myCar.Accelerate(55);

This has exactly the same effect as:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl23_code');" \o "Copy Code)

myCar.Accelerate(55, null, false);

It’s the same because the compiler will insert all the default values that you omit.

C# 4.0 will also let you call methods by specifying some arguments by name. In this way, you can pass an argument to an optional parameter without having to also pass arguments for all the parameters that come before it.

Say you want to call Accelerate to go in reverse, but you don’t want to specify the gear parameter. Well, you can do this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl24_code');" \o "Copy Code)

myCar.Accelerate(55, inReverse: true);

This is a new C# 4.0 syntax, and it’s the same as if you had written:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl25_code');" \o "Copy Code)

myCar.Accelerate(55, null, true);

In fact, whether or not parameters in the method you’re calling are optional, you can use names when passing arguments. For instance, these two calls are permissible and identical to one another:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl26_code');" \o "Copy Code)

Console.WriteLine(format: "{0:f}", arg0: 6.02214179e23);

Console.WriteLine(arg0: 6.02214179e23, format: "{0:f}");

If you’re calling a method that takes a long list of parameters, you can even use names as a sort of in-code documentation to help you remember which parameter is which.

On the surface, optional arguments and named parameters don’t look like interop features. You can use them without ever even thinking about interop. However, the motivation for these features comes from the Office APIs. Consider, for example, Word programming and something as simple as the SaveAs method on the Document interface. This method has 16 parameters, all of which are optional. With previous versions of C#, if you want to call this method you have to write code that looks like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl27_code');" \o "Copy Code)

Document d = new Document();

object filename = "Foo.docx";

object missing = Type.Missing;

d.SaveAs(ref filename, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing, ref missing);

Now, you can write this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl28_code');" \o "Copy Code)

Document d = new Document();

d.SaveAs(FileName: "Foo.docx");

I would say that’s an improvement for anyone who works with APIs like this. And improving the lives of programmers who need to write Office programs was definitely a motivating factor for adding named arguments and optional parameters to the language.

Now, when writing a .NET library and considering adding methods that have optional parameters, you’re faced with a choice. You can either add optional parameters or you can do what C# programmers have done for years: introduce overloads. In the Car.Accelerate example, the latter decision might lead you to produce a type that looks like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl29_code');" \o "Copy Code)

class Car {

public void Accelerate(uint speed) {

Accelerate(speed, null, false);

}

public void Accelerate(uint speed, int? gear) {

Accelerate(speed, gear, false);

}

public void Accelerate(uint speed, int? gear,

bool inReverse) {

/\* ... \*/

}

}

Selecting the model that suits the library you’re writing is up to you. Because C# hasn’t had optional parameters until now, the .NET Framework (including the .NET Framework 4) tends to use overloads. If you decide to mix and match overloads with optional parameters, the C# overload resolution has clear tie-breaking rules to determine which overload to call under any given circumstances.

**Indexed Properties**

Some smaller language features in C# 4.0 are supported only when writing code against a COM interop API. The Word interop in the previous illustration is one example.

C# code has always had the notion of an indexer that you can add to a class to effectively overload the [] operator on instances of that class. This sense of indexer is also called a default indexer, since it isn’t given a name and calling it requires no name. Some COM APIs also have indexers that aren’t default, which is to say that you can’t effectively call them simply by using []—you must specify a name. You can, alternatively, think of an indexed property as a property that takes some extra arguments.

C# 4.0 supports indexed properties on COM interop types. You can’t define types in C# that have indexed properties, but you can use them provided you’re doing so on a COM type. For an example of what C# code that does this looks like, consider the Range property on an Excel worksheet:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl30_code');" \o "Copy Code)

using Microsoft.Office.Interop.Excel;

class Program {

static void Main(string[] args) {

Application excel = new Application();

excel.Visible = true;

Worksheet ws =

excel.Workbooks.Add().Worksheets["Sheet1"];

// Range is an indexed property

ws.Range["A1", "C3"].Value = 123;

System.Console.ReadLine();

excel.Quit();

}

}

In this example, Range[“A1”, “C3”] isn’t a property called Range that returns a thing that can be indexed. It’s one call to a Range accessor that passes A1 and C3 with it. And although Value might not look like an indexed property, it, too, is one! All of its arguments are optional, and because it’s an indexed property, you omit them by not specifying them at all. Before the language supported indexed properties, you would have written the call like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl31_code');" \o "Copy Code)

ws.get\_Range("A1", "C3").Value2 = 123;

Here, Value2 is a property that was added simply because the indexed property Value wouldn’t work prior to C# 4.0.

**Omitting the Ref Keyword at COM Call Sites**

Some COM APIs were written with many parameters passed by reference, even when the implementation doesn’t write back to them. In the Office suite, Word stands out as an example—its COM APIs all do this.

When you’re confronted with such a library and you need to pass arguments by reference, you can no longer pass any expression that’s not a local variable or field, and that’s a big headache. In the Word SaveAs example, you can see this in action—you had to declare a local called filename and a local called missing just to call the SaveAs method, since those parameters needed to be passed by reference.

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl32_code');" \o "Copy Code)

Document d = new Document();

object filename = "Foo.docx";

object missing = Type.Missing;

d.SaveAs(ref filename, ref missing, // ...

You may have noticed in the new C# code that followed, I no longer declared a local for filename:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl33_code');" \o "Copy Code)

d.SaveAs(FileName: "Foo.docx");

This is possible because of the new omit ref feature for COM interop. Now, when calling a COM interop method, you can pass any argument by value instead of by reference. If you do, the compiler will create a temporary local on your behalf and pass that local by reference for you if required. Of course, you won’t be able to see the effect of the method call if the method mutates the argument—if you want that, pass the argument by ref.

This should make code that uses APIs like this much cleaner.

**Embedding COM Interop Types**

This is more of a C# compiler feature than a C# language feature, but now you can use a COM interop assembly without that assembly having to be present at run time. The goal is to reduce the burden of deploying COM interop assemblies with your application.

When COM interop was introduced in the original version of the .NET Framework, the notion of a Primary Interop Assembly (PIA) was created. This was an attempt to solve the problem of sharing COM objects among components. If you had different interop assemblies that defined an Excel Worksheet, we wouldn’t be able to share these Worksheets between components, because they would be different .NET types. The PIA fixed this by existing only once—all clients used it, and the .NET types always matched.

Though a fine idea on paper, in practice deploying a PIA turns out to be a headache, because there’s only one, and multiple applications could try to install or uninstall it. Matters are complicated because PIAs are often large, Office doesn’t deploy them with default Office installations, and users can circumvent this single assembly system easily just by using TLBIMP to create their own interop assembly.

So now, in an effort to fix this situation, two things have happened:

* identical COM interop types that share the same identifying characteristics (name, GUID and so on) as though they were actually the same .NET type.
* The C# compiler takes advantage of this by simply reproducing the interop types in your own assembly when you compile, removing the need for the interop assembly to exist at run time.

I have to omit some details in the interest of space, but even without knowledge of the details, this is another feature—like dynamic—you should be able to use without a problem. You tell the compiler to embed interop types for you in Visual Studio by setting the Embed Interop Types property on your reference to true.

Because the C# team expects this to be the preferred method of referencing COM assemblies, Visual Studio will set this property to True by default for any new interop reference added to a C# project. If you’re using the command-line compiler (csc.exe) to build your code, then to embed interop types you must reference the interop assembly in question using the /L switch rather than /R.

Each of the features I’ve covered in this article could itself generate much more discussion, and the topics all deserve articles of their own. I’ve omitted or glossed over many details, but I hope this serves as a good starting point for exploring C# 4.0 and you find time to investigate and make use of these features. And if you do, I hope you enjoy the benefits in productivity and program readability they were designed to give you.