New features in C# 4.0

Mads Torgersen, C# Language PM

Contents

[Introduction 1](#_Toc212875678)

[Dynamic binding 3](#_Toc212875679)

[Named and Optional Arguments 6](#_Toc212875680)

[Features for COM interop 8](#_Toc212875681)

[Variance 9](#_Toc212875682)

[Relationship with Visual Basic 11](#_Toc212875683)

[Resources 11](#_Toc212875684)

# Introduction

The major theme for C# 4.0 is dynamic programming. Increasingly, objects are “dynamic” in the sense that their structure and behavior is not captured by a static type, or at least not one that the compiler knows about when compiling your program. Some examples include

* objects from dynamic programming languages, such as Python or Ruby
* COM objects accessed through IDispatch
* ordinary .NET types accessed through reflection
* objects with changing structure, such as HTML DOM script objects
* data readers and other user defined dynamic objects

While C# remains a statically typed language, we aim to vastly improve the interaction with such objects.

A secondary theme is *co-evolution* with Visual Basic. Going forward we will aim to maintain the individual character of each language, but at the same time important new features should be introduced in both languages at the same time. They should be differentiated more by style and feel than by feature set.

The new features in C# 4.0 fall into four groups:

### Dynamic binding

Dynamic binding allows you to write method, operator and indexer calls, property and field accesses, and even object invocations which bypass the C# static type checking and instead gets resolved at runtime.

### Named and optional arguments

Parameters in C# can now be specified as optional by providing a default value for them in a member declaration. When the member is invoked, optional arguments can be omitted. Furthermore, any argument can be passed by parameter name instead of position.

### COM specific interop features

Dynamic binding as well as named and optional arguments help making programming against COM less painful than today. On top of that, however, we are adding a number of other features that further improve the interop experience specifically with COM.

### Variance

It used to be that an IEnumerable<string> wasn’t an IEnumerable<object>. Now it is – C# embraces type safe “co-and contravariance,” and common BCL types are updated to take advantage of that.

# Dynamic Binding

Dynamic binding offers a unified approach to invoking things dynamically. With dynamic binding, when you have an object in your hand, you do not need to worry about whether it comes from COM, IronPython, the HTML DOM, reflection or elsewhere; you just apply operations to it and leave it to the runtime to figure out what exactly those operations mean for that particular object.

This affords you enormous flexibility, and can greatly simplify your code, but it does come with a significant drawback: Static typing is not enforced for these operations. A dynamic object is assumed at compile time to support any operation, and only at runtime will you get an error if it wasn’t so. Oftentimes this will be no loss, because the object wouldn’t have a static type anyway, in other cases it is a tradeoff between brevity and safety. In order to facilitate this tradeoff, it is a design goal of C# to allow you to opt in or opt out of dynamic behavior on every single call.

## The dynamic type

C# 4.0 introduces a new static type called dynamic. When you have an object of type dynamic you can “do things to it” that are resolved only at runtime:

dynamic d = GetDynamicObject(…);  
d.M(7);

The C# compiler allows you to call a method with any name and any arguments on d because it is of type dynamic. At runtime the actual object that d refers to will be examined to determine what it means to “call M with an int” on it.

The type dynamic can be thought of as a special version of the type object, which signals that the object can be used dynamically. It is easy to opt in or out of dynamic behavior: any object can be implicitly converted to dynamic, “suspending belief” until runtime. Conversely, expressions of type dynamic can be implicitly converted to object, or indeed any other type, as long as there exists a conversion at runtime:

dynamic d = 7; // compile-time implicit conversion  
int i = d; // runtime implicit conversion

## Dynamic operations

Not only method calls, but also field and property accesses, indexer and operator calls and even delegate invocations and constructor calls can be dispatched dynamically:

dynamic d = GetDynamicObject(…);  
d.M(7); // calling methods  
d.f = d.P; // getting and settings fields and properties  
d[“one”] = d[“two”]; // getting and setting through indexers  
int i = d + 3; // calling operators  
string s = d(5,7); // invoking as a delegate  
var c = new C(d); // calling a constructor

The role of the C# compiler here is simply to package up the necessary information about “what is being done to d”, so that the runtime can pick it up and determine what the exact meaning of it is given an actual object d. Think of it as deferring part of the compiler’s job to runtime.

The result of any dynamic operation is itself of type dynamic, with two exceptions:

* The type of a dynamic constructor call is the constructed type
* The type of a dynamic implicit or explicit conversion is the target type of the conversion.

## Runtime lookup

At runtime a dynamic operation is dispatched according to the nature of its target object d:

### Dynamic objects

If d implements the interface IDynamicMetaObjectProvider, it is a so-called *dynamic object*, which means that it will itself be asked to bind and perform the operation. Thus by implementing IDynamicMetaObjectProvider a type can completely redefine the meaning of operations such as method calls, member access etc. This is used intensively by dynamic languages such as IronPython and IronRuby to implement their own dynamic object models. It is also used by APIs, e.g. by the Silverlight HTML DOM to allow direct access to the object’s properties and methods using member access and method call syntax instead of string-based accessor methods such as SetProperty or Invoke.

### COM objects

If d is a COM object, the operation is dispatched dynamically through COM IDispatch. This allows calling to COM types that don’t have a Primary Interop Assembly (PIA), and relying on COM features that don’t have a counterpart in C#, such as default properties.

### Plain objects

Otherwise d is a standard .NET object, and the operation will be dispatched using reflection on its type and a C# “runtime binder” which implements C#’s lookup and overload resolution semantics at runtime. This is essentially a part of the C# compiler running as a runtime component to “finish the work” on dynamic operations that was deferred by the static compiler.

## Example

Assume the following code:

dynamic d1 = new Foo();  
dynamic d2 = new Bar();  
string s;

d1.M(s, d2, 3, null);

Because the receiver and an argument of the call to M are dynamic, the C# compiler does not try to resolve the meaning of the call. Instead it stashes away information for the runtime about the call. This information (often referred to as the “payload”) is essentially equivalent to:

“Perform an instance method call of a method called M with the following arguments:

1. a string
2. a dynamic
3. a literal int 3
4. a literal object null”

At runtime, assume that the actual type Foo of d1 is not a dynamic object. In this case the C# runtime binder picks up to finish the overload resolution job based on runtime type information, proceeding as follows:

1. Reflection is used to obtain the actual runtime types of the two objects, d1 and d2, that did not have a static type (or rather had the static type dynamic). The result is Foo for d1 and Bar for d2.
2. Method lookup and overload resolution is performed on the type Foo with the call M(string,Bar,3,null) using ordinary C# semantics.
3. If the method is found it is invoked; otherwise a runtime exception is thrown.

## Overload resolution with dynamic arguments

Even if the receiver of a method call is of a static type, overload resolution can still happen at runtime. This will happen if one or more of the arguments have the type dynamic:

Foo foo = new Foo();  
dynamic d = new Bar();

var result = foo.M(d);

The C# runtime binder will choose between the statically known overloads of M on Foo, based on the *runtime* type of d, namely Bar. The result is again of type dynamic.

## The Dynamic Language Runtime

An important component in the underlying implementation of dynamic binding is the Dynamic Language Runtime (DLR), which is a new API in .NET 4.0.

The DLR provides most of the infrastructure behind not only C# dynamic binding but also the implementation of several dynamic programming languages on .NET, such as IronPython and IronRuby. Through this common infrastructure a high degree of interoperability is ensured, but just as importantly the DLR provides excellent caching mechanisms which serve to greatly enhance the efficiency of runtime dispatch.

To the user of dynamic binding in C#, the DLR is invisible except for the improved efficiency. However, if you want to implement your own dynamically dispatched objects, the IDynamicMetaObjectProvider interface allows you to interoperate with the DLR and plug in your own behavior. Doing this directly is a rather advanced task, which requires you to understand a good deal more about the inner workings of the DLR. Fortunately .NET 4.0 provides several helper classes to make this task a lot easier, and for API writers, it can definitely be worth the trouble as you can sometimes vastly improve the usability of libraries representing an inherently dynamic domain.

## Limitations

There are a few limitations and things that might work differently than you would expect.

* The DLR allows objects to be created from objects that represent classes. However, the current implementation of C# doesn’t have syntax to support this.
* Dynamic binding will not be able to find extension methods. Whether extension methods apply or not depends on the static context of the call (i.e. which using clauses occur), and this context information is not kept as part of the payload.
* Anonymous functions (i.e. lambda expressions) cannot appear as arguments to a dynamic operation. The compiler cannot bind (i.e. “understand”) an anonymous function without knowing what type it is converted to.

One consequence of these limitations is that you cannot easily use LINQ queries over dynamic objects:

dynamic collection = …;

var result = collection.Select(e => e + 5);

If the Select method is an extension method, dynamic binding will not find it. Even if it is an instance method, the above does not compile, because a lambda expression cannot be passed as an argument to a dynamic operation.

# Named Arguments and Optional Parameters

Named arguments and optional parameters are really two distinct features, but are often useful together. Optional parameters allow you to omit arguments to member invocations, whereas named arguments is a way to provide an argument using the name of the corresponding parameter instead of relying on its position in the parameter list.

Some APIs, most notably COM interfaces such as the Office automation APIs, are written specifically with named and optional parameters in mind. Up until now it has been very painful to call into these APIs from C#, with sometimes as many as thirty arguments having to be explicitly passed, most of which have reasonable default values and could be omitted.

Even in APIs for .NET however you sometimes find yourself compelled to write many overloads of a method with different combinations of parameters, in order to provide maximum usability to the callers. Optional parameters are a useful alternative for these situations.

## Optional parameters

A parameter is declared optional simply by providing a default value for it:

public void M(int x, int y = 5, int z = 7);

Here y and z are optional parameters and can be omitted in calls:

M(1, 2, 3); // ordinary call of M  
M(1, 2); // omitting z – equivalent to M(1, 2, 7)  
M(1); // omitting both y and z – equivalent to M(1, 5, 7)

Default argument values are somewhat restricted. They must be given as constant expressions, or default value expressions default(T).

## Named and optional arguments

C# 4.0 does not permit you to omit arguments between commas as in M(1,,3). This could lead to highly unreadable comma-counting code. Instead if you want to omit arguments in the middle, any argument can be passed by name. Thus if you want to omit only y from a call of M you can write:

M(1, z: 3); // passing z by name

or

M(x: 1, z: 3); // passing both x and z by name

or even

M(z: 3, x: 1); // reversing the order of arguments

All forms are equivalent, except that arguments are always evaluated in the order they appear, so in the last example the 3 is evaluated before the 1.

Optional and named arguments can be used not only with methods but also with indexers and constructors.

## Overload resolution

Named and optional arguments affect overload resolution, but the changes are relatively simple:

A signature is *applicable* if all its parameters are either optional or have exactly one corresponding argument (by name or position) in the call which is convertible to the parameter type.

Betterness rules on conversions are only applied for arguments that are explicitly given – omitted optional arguments are ignored for betterness purposes.

If two signatures are equally good, one that does not omit optional parameters is preferred.

M(string s, int i = 1);  
M(object o);  
M(int i, string s = “Hello”);  
M(int i);

M(5);

Given these overloads, we can see the working of the rules above. M(string,int) is not applicable because 5 doesn’t convert to string. M(int,string) is applicable because its second parameter is optional, and so, obviously are M(object) and M(int).

M(int,string) and M(int) are both better than M(object) because the conversion from 5 to int is better than the conversion from 5 to object.

Finally M(int) is better than M(int,string) because no optional arguments are omitted.

Thus the method that gets called is M(int).

# Features for COM interop

Dynamic binding as well as named and optional parameters greatly improve the experience of interoperating with COM APIs such as the Office Automation APIs. In order to remove the remaining speed bumps, a couple of small COM-specific features are also added to C# 4.0.

## Compiling without PIAs

Primary Interop Assemblies are large .NET assemblies generated from COM interfaces to facilitate strongly typed interoperability. They provide great support at design time, where your experience of the interop is as good as if the types where really defined in .NET. However, at runtime these large assemblies can easily bloat your program, and also cause versioning issues because they are distributed independently of your application.

The embedded-PIA feature allows you to continue to use PIAs at design time without having them around at runtime. Instead, the C# compiler will bake the small part of the PIA that a program actually uses directly into its assembly. At runtime the PIA does not have to be loaded.

## Dynamic import

Many COM methods accept and return “variant” types, which are represented in the PIAs as object. In the vast majority of cases, a programmer calling these methods already knows the static type of a returned object from context, but explicitly has to perform a cast on the returned value to make use of that knowledge. These casts are so common that they constitute a major nuisance.

In order to facilitate a smoother experience, if you choose to import these COM APIs with PIA-embedding, variants are instead represented using the type dynamic. In other words, from your point of view, COM signatures now have occurrences of dynamic instead of object in them.

This means that you can easily access members directly off a returned object, or you can assign it to a strongly typed local variable without having to cast. To illustrate, you can now say

excel.Cells[1, 1].Value = "Hello";

instead of

((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello";

and

Excel.Range range = excel.Cells[1, 1];

instead of

Excel.Range range = (Excel.Range)excel.Cells[1, 1];

## Omitting ref

Because of a different programming model, many COM APIs contain a lot of reference parameters. Contrary to refs in C#, these are typically not meant to mutate a passed-in argument for the subsequent benefit of the caller, but are simply another way of passing value parameters.

It therefore feels unreasonable to a C# programmer to have to create temporary variables for all such ref parameters and pass these by reference. Instead, specifically for COM methods, the C# compiler will allow you to pass arguments by value to such reference parameters, and will automatically generate temporary variables to hold the passed-in values, subsequently discarding these when the call returns. In this way the caller sees value semantics, and will not experience any side effects, but the called method still gets a reference.

## Indexed properties

Many COM APIs expose “indexed properties” which are essentially properties with parameters. C# will not allow you to declare indexed properties, but to the extent that non-C# APIs expose them, will now allow you to access these using element access syntax. So instead of

o.set\_P(i+1, o.get\_P(i) \* 2);

You can now write the more intuitive

o.P[i+1] = o.P[i] \* 2;

## Limitations

A few COM interface features still are not surfaced in C#, most notably default properties. As mentioned above these will be respected if you access COM dynamically, but statically typed C# code will still not recognize them.

## Larger COM Example

Here is a larger Office automation example that shows many of the new C# features in action.

using System;  
using System.Diagnostics;  
using System.Linq;  
using Excel = Microsoft.Office.Interop.Excel;  
using Word = Microsoft.Office.Interop.Word;

class Program  
{  
 static void Main(string[] args) {  
 var excel = new Excel.Application();  
 excel.Visible = true;

excel.Workbooks.Add(); // optional arguments omitted

excel.Cells[1, 1].Value = "Process Name"; // no casts; Value dynamically   
 excel.Cells[1, 2].Value = "Memory Usage"; // accessed

var processes = Process.GetProcesses()  
 .OrderByDescending(p =&gt; p.WorkingSet)  
 .Take(10);

int i = 2;  
 foreach (var p in processes) {  
 excel.Cells[i, 1].Value = p.ProcessName; // no casts  
 excel.Cells[i, 2].Value = p.WorkingSet; // no casts  
 i++;  
 }

Excel.Range range = excel.Cells[1, 1]; // no casts

Excel.Chart chart = excel.ActiveWorkbook.Charts.  
 Add(After: excel.ActiveSheet); // named and optional arguments

chart.ChartWizard(  
 Source: range.CurrentRegion,  
 Title: "Memory Usage in " + Environment.MachineName); //named+optional

chart.ChartStyle = 45;

chart.CopyPicture(Excel.XlPictureAppearance.xlScreen,  
 Excel.XlCopyPictureFormat.xlBitmap,  
 Excel.XlPictureAppearance.xlScreen);

var word = new Word.Application();  
 word.Visible = true;

word.Documents.Add(); // optional arguments

word.Selection.Paste();  
 }  
}

The code is much more terse and readable than the C# 3.0 counterpart.

# Variance

An aspect of generics that often comes across as surprising is that the following is illegal:

IList<string> strings = new List<string>();  
IList<object> objects = strings;

The second assignment is disallowed because strings does not have the same element type as objects. There is a perfectly good reason for this. If it were allowed you could write:

objects[0] = 5;  
string s = strings[0];

Allowing an int to be inserted into a list of strings and subsequently extracted as a string. This would be a breach of type safety.

However, there are certain interfaces where the above cannot occur, notably where there is no way to insert an object into the collection. Such an interface is IEnumerable<T>. If instead you say:

IEnumerable<object> objects = strings;

Things are a lot safer: There is no way we can put the wrong kind of thing into strings through objects, because objects doesn’t *have* a method that takes an element as input. Variance is about allowing assignments such as this in cases where it is safe. The result is that a lot of situations that were previously surprising now just work.

## Covariance

In .NET 4.0 the IEnumerable<T> interface will be declared in the following way:

public interface IEnumerable<out T> : IEnumerable  
{  
 IEnumerator<T> GetEnumerator();  
}

public interface IEnumerator<out T> : IEnumerator  
{  
 bool MoveNext();  
 T Current { get; }  
}

The “out” in these declarations is a new C# 4.0 modifier which signifies that the T can only occur in output position in the interface – the compiler will complain otherwise. In return for this restriction, the interface becomes “covariant” in T, which means that an IEnumerable<A> is considered an IEnumerable<B> if A has a reference conversion to B.

As a result, any sequence of strings is also e.g. a sequence of objects.

This is useful e.g. in many LINQ methods. Using the declarations above:

var result = strings.Union(objects); // succeeds with an IEnumerable<object>

This would previously have been disallowed, and you would have had to to some cumbersome wrapping to get the two sequences to have the same element type.

## Contravariance

Type parameters can also have an “in” modifier, restricting them to occur only in input positions. An example is IComparer<T>:

public interface IComparer<in T>  
{  
 public int Compare(T left, T right);  
}

The somewhat baffling result is that an IComparer<object> can in fact be considered an IComparer<string>! It makes sense when you think about it: If a comparer can compare *any* two objects, it can certainly also compare two strings. So a comparer of objects is a comparer of strings. This property is referred to as contravariance.

A generic type can have both in and out modifiers on its type parameters, as is the case with the Func<…> delegate types:

public delegate TResult Func<in TArg, out TResult>(TArg arg);

Obviously the argument only ever comes *in*, and the result only ever comes *out*. Therefore a Func<object,string> can in fact be used as a Func<string,object>.

## Limitations

Variant type parameters can only be declared on interfaces and delegate types, due to a restriction in the CLR. Variance only applies when there is a *reference* conversion between the type arguments. For instance, an IEnumerable<int> is not an IEnumerable<object> because the conversion from int to object is a boxing conversion, not a reference conversion.

# Relationship with Visual Basic

A number of the features introduced to C# 4.0 already exist or will be introduced in some form or other in Visual Basic:

* Late binding in VB is similar in many ways to dynamic binding in C#. In VB 10 (the “sister” VB version to C# 4.0), late binding has been extended to target the DLR for dynamic objects. Thus VB has the same degree of integration with dynamic objects as does C#.
* Named and optional arguments have been part of Visual Basic for a long time, and the C# version of the feature is explicitly engineered with maximal VB interoperability in mind.
* VB also already allows reference parameters to be omitted, and exposes indexed properties.
* PIA embedding and variance are both being introduced to VB and C# at the same time.

VB in turn is adding a number of features that have hitherto been a mainstay of C#. As a result future versions of C# and VB will have much better feature parity, for the benefit of everyone.