C#

**Version 4.0 Specification**

**Draft – March 2009**

**Notice**

© 1999-2009 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Other product and company names mentioned herein may be the trademarks of their respective owners.

**Table of Contents**

19. Introduction to C# 4.0 1

19.1 Dynamic binding 1

19.2 Named and optional parameters 1

19.3 COM specific interop features 1

19.4 Variance 1

19.5 About this document 1

20. Dynamic Binding 2

20.1 The dynamic type 3

20.2 Dynamic binding 4

20.3 Compile time semantics of dynamic binding 4

20.3.1 Static binding with dynamic arguments 5

20.3.2 Dynamic binding with a statically known candidate set 5

20.3.3 Conversion to interface types 5

20.3.4 Dynamic collections in foreach statements 5

20.3.5 Dynamic resources in using statements 6

20.3.6 Compound operators 6

20.4 Runtime semantics of dynamic binding 7

21. Named and Optional Arguments 8

21.1 Optional arguments 8

21.2 Named Arguments 9

21.3 Overload resolution 10

21.4 Better function 10

21.5 Invocation 11

22. COM Interoperability 12

22.1 Passing values to reference parameters 12

22.2 Linking of Primary Interop Assemblies 13

22.3 Dynamification of Linked COM members 13

22.4 The COM runtime binder 13

22.5 Example 13

23. Co- and Contravariance 15

23.1 Covariance 15

23.2 Contravariance 15

23.3 Limitations 16

23.4 Syntax 16

23.5 Variance safety 16

23.6 Interface declarations 17

23.6.1 Interface methods 17

23.6.2 Other interface members 18

23.7 Delegate declarations 18

23.8 Conversions 18

23.8.1 Implicit conversions 18

23.8.2 Explicit conversions 19

23.9 Type Inference 19

23.9.1 The first phase 19

23.9.2 Exact inferences 19

23.9.3 Lower-bound inferences 19

23.9.4 Upper-bound inferences 20

23.9.5 Fixing 21

# Introduction to C# 4.0

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 objects

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

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

## Dynamic binding

Dynamic lookup allows you to write method, operator and indexer calls, property and field accesses, and even object invocations which bypass the normal static binding of C# and instead gets resolved dynamically.

## Named and optional parameters

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 lookup as well as named and optional parameters both help making programming against COM types less painful than today. On top of that, however, a number of other small features further improve the interop experience.

## Variance

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

## About this document

This is an early draft of the specification of the new features in C# 4.0. Eventually the specification of the new language features will be merged into the main document, which currently describes all of C# 3.0. However, for an initial period of time before the actual product release of the new version it is more useful to keep the specification of the new features separate.

Please be patient and understanding about the fact that this document is not yet in its final state. It is a best effort to describe the features as they are intended to work at the time of writing, to the largest degree of detail possible, but it will undoubtedly have many shortcomings both in detail, accuracy and textual quality.

# Dynamic Binding

Dynamic binding provides a unified approach to selecting opeartions dynamically. With dynamic binding developer does not need to worry about whether a given object comes from e.g. COM, IronPython, the HTML DOM or reflection; operations can uniformly be applied to it and the runtime will determine what those operations mean for that particular object.

This affords enormous flexibility, and can greatly simplify the source code, but it does come with a significant drawback: Static typing is not maintained for these operations. A dynamic object is assumed at compile time to support any operation, and only at runtime will an error occur if it was not so.

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);

C# 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, the compiler allows implicit conversion of dynamic expressions to any type:

dynamic d = 7; // statically bound implicit conversion

int i = d; // dynamically bound implicit conversion

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

dynamic d = GetDynamicObject(…);

d.M(7); // calling methods  
d.M(x: “Hello”); // passing arguments by name  
d.f = d.P; // getting and setting 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  
C c = new C(d); // selecting constructors

The role of the C# compiler 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 referenced by d. Think of it as deferring part of the compiler’s job to runtime.

The result of most dynamic operations is itself of type dynamic. The exceptions are conversions and constructor invocations, both of which have a natural static type.

At runtime a dynamic operation is resolved according to the nature of its target object d: If d implements the special interface IDynamicObject, d itself is asked to perform the operation. Thus by implementing IDynamicObject a type can completely redefine the meaning of dynamic operations. This is used intensively by dynamic programming languages such as IronPython and IronRuby to implement their own dynamic object models. It can also be used by APIs, e.g. to allow direct access to the object’s dynamic properties using property syntax.

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

Assume the following code:

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

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

Because the receiver of the call to M is 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 M with the following arguments:

* a string
* a dynamic
* a literal int 3
* a literal object null”

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

* 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.
* Method lookup and overload resolution is performed on the type Foo with the call M(string,Bar,3,null) using ordinary C# semantics.
* If the method is found it is invoked; otherwise a runtime exception is thrown.

## The dynamic type

The grammar is extended with the following type expression:

type:  
...  
dynamic

The types dynamic and object are considered the same, and are semantically equivalent in every way except the following two cases:

* Expressions of type dynamic can cause dynamic binding to occur in specific situations
* Type inference algorithms as described in §7.4 will prefer dynamic over object if both are candidates

This deep equivalence means for instance that:

* There is an implicit identity conversion between object and dynamic
* There is an implicit identity conversion between constructed types that differ only by dynamic versus object
* Method signatures that differ only by dynamic versus object are considered the same
* Like with object, there is an implicit conversion from every type (other than pointer types) to dynamic and an explicit conversion from dynamic to every such type.

The type dynamic does not have a separate runtime representation from object – for instance the expression

typeof(dynamic) == typeof(object)

is true.

An expression of the type dynamic is referred to as a dynamic expression.

## Dynamic binding

The purpose of the dynamic type is to affect the way operations are selected by the compiler. The process of selecting which operation to apply based on the types of constituent expressions is referred to as binding.

The following operations in C# are selected based on some form of binding:

* Member access: e.M
* Method invocation: e.M(e1,…,en)
* Delegate invocaton: e(e1,…,en)
* Element access: e[e1,…,en]
* Constructor calls: new C(e1,…,en)
* Overloaded unary operators: +, -, !, ~, ++, --, true, false
* Overloaded binary operators: +, -, \*, /, %, &, &&, |, ||, ??, ^, <<, >>, ==,!=, >, <, >=, <=
* Compound assignment operators: +=, -=, etc.
* Implicit and explicit conversions

When dynamic expressions are not involved, C# always defaults to static binding, which means that the compile-time types of constituent expressions are used in the selection process. However, when one of the constituent expressions in the above listed operations is a dynamic expression, the operation is instead dynamically bound.

## Compile time semantics of dynamic binding

An anonymous function cannot be used as a constituent value of a dynamically bound operation.

A method group can only be a constituent value of a dynamically bound operation if it is immediately invoked.

Other than that, a dynamically bound operation always succeeds at compile time, unless otherwise specified in the following.

The static result type of most dynamically bound operations is dynamic. The only exceptions are:

* Conversions, which has the static type that is being converted to
* Constructor invocations which have the static type that is being constructed

Most dynamically bound operations are classified as a value. The only exceptions are member accesses and element accesses, which are classified as variables. However, they can not be used as arguments to ref or out parameters.

### Static binding with dynamic arguments

In certain cases if enough is known statically, the above operations will not lead to dynamic binding. This is the case for:

* Element accesses where the static type of the receiver is an array type
* Delegate invocations where the static type of the delegate is a delegate type

In these cases the operation is resolved statically, instead implicitly converting dynamic arguments to their required type. Thus, the result type of such operations is statically known.

### Dynamic binding with a statically known candidate set

For most dynamically bound operations the set of possible candidates for resolution is unknown at compile time. In certain cases, however the candidate set is known:

* Static method calls with dynamic arguments
* Instance method calls where the static type of the receiver is not dynamic
* Indexer calls where the static type of the receiver is not dynamic
* Constructor calls

In these cases a limited compile time check is performed for each candidate to see if any of them could possibly apply at runtime. This check includes:

* Checking that the candidate has the right name and arity
* Performing a partial type inference to check that inferences do exist where not depending on arguments with the static type dynamic
* Checking that any arguments not statically typed as dynamic match parameter types that are known at compile time

If no candidate passes this test, a compile time error occurs.

Extension methods are not considered candidates for instance method calls, because they are not available during runtime binding.

### Conversion to interface types

In C# user defined conversions to interface types are not allowed. For performance purposes conversions of dynamic expressions to interface types are therefore statically rather than dynamically bound. This does have a slight semantic effect, because a dynamic object (i.e. an object whose runtime type implements IDynamicObject) could have given other meaning to the conversion, had it been dynamically bound.

In foreach and using statements (§20.3.4 and §20.3.5), expansions may do dynamic casts to interfaces even though that cannot be done directly from source code.

### Dynamic collections in foreach statements

If the collection expression of a foreach statement (§8.8.4) has the static type dynamic, then the collection type is System.IEnumerable, the enumerator type is the interface System.Collections.IEnumerator, and the element type is object.

Furthermore the cast that is part of the foreach expansion is bound dynamically, not statically as is otherwise the case for interface types (§20.3.3). This enables dynamic objects to participate in a foreach loop even if they do not implement the IEnumerable interface directly.

### Dynamic resources in using statements

A using statement (§8.13) is allowed to have a dynamic expression as the resource acquisition. More specifically, if the form of resource-acquisition is expression and the type of the expression is dynamic, or if the form of resource-acquisition is local-variable-declaration and the type of the local-variable-declaration is dynamic, then the using statement is allowed.

In this case, the conversion of the expression or local variables to IDisposible occurs before the body of the using statement is executed, to ensure that the conversion does in fact succeed. Furthermore the conversion is bound dynamically, not statically as is otherwise the case for conversion of dynamic to interface types (§20.3.3). This enables dynamic objects to participate in a using statement even if they do not implement the IDisposable interface directly.

A using statement of the form

using (expression) statement

where the type of expression is dynamic, is expanded as:

{ IDisposable \_\_d = (IDisposable)expression // as a dynamic cast  
 try {  
 statement;  
 }  
 finally {  
 if (\_\_d != null) \_\_d.Dispose();  
 }  
}

A using statement of the form

using (dynamic resource = expression) statement

is expanded as:

{  
 dynamic resource = expression;  
 IDisposable \_\_d = (IDisposable)resource // as a dynamic cast  
 try {  
 statement;  
 }  
 finally {  
 if (\_\_d != null) \_\_d.Dispose();  
 }  
}

In either expansion, the resource variable is read-only and the \_\_d variable is invisible in the embedded statement. Also, as already mentioned, the cast to IDisposable is bound dynamically.

### Compound operators

Compound operators x *binop*= y are bound as if expanded to the form x = x *binop* y, but both the *binop* and assignment operations, if bound dynamically, are specially marked as coming from a compound assignment.

At runtime if all the following are true:

* x is of the form d.X where d is of type dynamic
* the runtime type of d declares X to have type T or T? where T is a primitive type
* the result of x *binop* y has the runtime type S where S is a primitive type
* S and T are either both integral types (sbyte, byte, short, ushort, int, uint, long or ulong) or both floating point types (float or double)
* S is implicitly convertible to T

Then the result of x *binop* y is explicitly converted to T before being assigned. This is to mimic the corresponding behavior of statically bound compound assignments, which will explicitly convert the result of a primitive *binop* to the type of the left hand side variable (§7.16.2).

For dynamically bound += and -= the compiler will emit a call to check dynamically whether the left hand side of the += or -= operator is an event. If so, the dynamic operation will be resolved as an event subscription or unsubscription instead of through the expansion above.

## Runtime semantics of dynamic binding

Unless specified otherwise in the following, dynamic binding is performed at runtime and generally proceeds as follows:

* If the receiver is a dynamic object – i.e., implements an implementation-specific interface that we shall refer to as IDynamicObject – the object itself programmatically defines the resolution of the operations performed on it.
* Otherwise the operation gets resolved at runtime in the same way as it would have at compile time, using the runtime type of any constituent value statically typed as dynamic and the compile time type of any other constituent value.
* If a constituent value derives from a literal, the dynamic binding is able to take that into account. For instance, some conversions are available only on literals.
* If a constituent value of static type dynamic has the runtime value null, it will be treated as if the literal null was used.
* Extension method invocations will not be considered – the set of available extension methods at the site of the call is not preserved for the runtime binding to use.
* If the runtime binding of the operation succeeds, the operation is immediately performed,otherwise a runtime error occurs.

In reality the runtime binder will make heavy use of caching techniques in order to avoid the performance overhead of binding on each call. However, the observed behavior is the same as described here.

# Named and Optional Arguments

Named 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.

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)

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 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.

## Optional arguments

Formal parameters of constructors, methods, indexers and delegate types can be declared optional:

fixed-parameter:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

A fixed-parameter with a default-argument is an optional parameter, whereas a fixed-parameter without a default-argument is a required parameter.

A required parameter cannot appear after an optional parameter in a formal-parameter-list.

A ref or out parameter cannot have a default-argument.

The expression in a default-argument must be one of the following:

* a constant-expression
* an expression of the form new S() where S is a value type
* an expression of the form default(S) where S is a value type

The expression must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

Note that the grammar permits a parameter-array to occur after an optional parameter, but prevents a parameter-array from having a default value – the omission of arguments for a parameter-array would instead result in the creation of an empty array.

Example:

public struct T  
{  
 public void M(  
 int i,  
 bool b = false,  
 bool? n = false,  
 string s = "Hello",  
 object o = null,  
 T t = default(T))  
 {  
 …  
 }  
}

Here i is a required parameter and b, s, o and t are optional parameters.

If certain cases where a default argument is specified that can never be applied, the compiler must yield a warning. This is the case for

* partial method definitions, because only default arguments specified in the declaration are used
* operator declarations (including conversions), because the application syntax does not allow operands to be omitted
* explicit interface member implementations, because they are never called directly
* single-parameter indexer declarations, because indexer access must always have at least one argument.

A function member with optional parameters may be invoked without explicitly passing arguments for those parameters. For arguments that are passed implicitly, the default arguments specified in the declaration of the parameter are used instead.

## Named Arguments

A function member may be invoked with named as well as positional arguments.

argument:  
argument-nameopt argument-value

argument-name:  
identifier :

argument-value:  
expression  
ref variable-reference  
out variable-reference

An argument with an argument-name is a named argument. An argument without an argument-name is a positional argument.

Indexers are changed to have argument-lists instead of expression-lists:

element-access:  
primary-no-array-creation-expression [ argument-list ]

The argument-list of an element-access is not allowed to contain ref or out arguments. Furthermore, the argument-list of an array access is not allowed to contain named arguments.

Attributes allow named arguments in this style, as well as the attribute-style named arguments used to initialize properties of the attribute:

positional-argument:  
argument-nameopt attribute-argument-expression

It is an error for a positional argument to appear after a named argument.

Note that the syntax changes do not allow for the omission of positional arguments in the middle of an argument list; i.e. M(7,,false) is not syntactically valid.

## Overload resolution

When determining the applicability of a function member the list of arguments are mapped to the formal parameters of the function member. The positional arguments are mapped to the parameters in the corresponding positions, and the named arguments are mapped to the parameters with the corresponding names.

For virtual and abstract members, the parameter names in overrides may be different from those in the declaration. For the purposes of overload resolution, the parameter names that apply are the ones that appear in the *most specific* override of the function member with respect to the static type of the target of the member access.

For partial methods, the parameter names and default arguments used are those of the declaring method. Thus, the presence or absence of a corresponding definition of the partial method does not influence overload resolution.

If each argument does not map to a separate parameter in this fashion, the function member is not applicable.

If no argument maps to a given required parameter, the function member is not applicable.

Otherwise, a positional argument list is constructed by placing every named argument in the position of its corresponding parameter, and by supplying for each parameter for which an argument was not explicitly passed, the corresponding default argument.

The function member is applicable if it is applicable by the rules of §7.4.3.1 with respect to the positional argument list thus constructed.

## Better function

The “better conversion” rules in the overload resolution specification only apply where arguments are actually given. Thus, optional parameters for which no argument is passed are ignored for the purposes of conversion betterness.

Also, note that betterness is specified per *argument* not per *parameter*. Thus the betterness rules check whether the conversion of a given argument to the parameter type that it corresponds to in one member, is better than the conversion to the parameter type it corresponds to in the other – per the mapping of arguments described in §21.3.

As a tie breaker rule, a function member for which all arguments where explicitly given is better than one for which default values were supplied in lieu of explicit arguments. This tie breaker rule should appear last. In particular it occurs after the tie breaker rule preferring methods that haven’t been expanded for params arguments. This means that params methods get rejected first, and optional parameters hence win over expanded params parameters.

## Invocation

The expressions in the argument-list are evaluated in the order they appear in the argument-list, and are then passed to the invocation in the order described by the positional argument list constructed during overload resolution (§21.3).

# COM Interoperability

Interoperability with COM on the Microsoft .NET platform from C# is traditionally a painful experience. The optional and named arguments features of C# 4.0 do a lot to alleviate this, but there are still some painpoints, most of which are addressed by the features in this section.

It is important to note that these features are Microsoft specific extensions to the C# language. Implementation of these features is not required to be a conformant C# 4.0 implementation.

Common to these features is that they work only with members of “COM types”, i.e. types that implement a GUID attribute. In the following, the term “COM method” refers to a method of a COM type.

## Passing values to reference parameters

In C# reference parameters a generally used only when a method intends to modify the contents of the passed-in variable. To ensure that a caller is aware of this potential mutation, C# requires explicit use of the ref keyword when such a method is called.

However, in COM a different pattern prevails: A COM method may well have reference parameters simply because of a perceived performance benefit in parameter passing over value parameters. In the common case a COM method will not modify its parameters even when passed by reference. It therefore seems unnecessarily inhibitive that a caller of such a method from C# should have to declare temporary variables for all these arguments, and pass those by reference.

For this reason calls to COM methods are allowed to pass arguments by value (i.e. without the ref keyword) even when the method signature indicates a reference parameter. The semantics are as follows:

* A temporary variable of the appropriate type is allocated by the compiler
* The value of the argument is assigned into the temporary variable
* The temporary variable is passed to the method by reference
* Upon return, the temporary variable is discarded – any modifications to it caused by the called method will not be observed.

This does not in any way change the evaluation order of arguments.

As an example, given a COM method with the following signature:

void M(int i, ref int r1, ref int r2, ref int r3)

And a call:

int x = 0;  
M(1, 2, x, ref x);

The first argument 1 is passed by value. For the second argument 2 a temporary variable is created holding that value and passed by reference to the call. For the third argument x, even though the expression x is a variable, it is reclassified as a value because the ref modifier is not used. Therefore, a temporary variable is created, the value of x – zero – is assigned to it, and the variable is passed by reference to the call. Thus any modifications to the value of r1 and r2 inside of the method are discarded upon return along with the temporary variables created for them. The fourth argument, ref x, is passed in the normal fashion, by reference to the variable x. Any modification to r3 in the method body will be reflected in the variable x upon return.

## Linking of Primary Interop Assemblies

Primary Interop Assemblies (PIAs) are .NET libraries wrapping COM types for calling from .NET code. Traditionally, COM-calling C# code is compiled against the PIAs, and at runtime the PIAs in the execution environment will be loaded to facilitate the calls to the wrapped COM functionality.

In C# 4.0 PIAs can be “linked” instead of “referenced”. The significance is that any part of the PIA that is called from the client program will be copied into the client assembly itself. At runtime, therefore, there is no need to load and consult a PIA in the execution environment.

There are a couple of benefits to this approach: First of all the size of the running program can often be drastically reduced, because PIAs tend to be big, and only small parts of them tend to be called from any given program. Moreover, the versioning issues that might arise from differences between the PIAs on the compilation and execution machine are avoided.

The use of linking instead of referencing is for the most part semantically transparent. For instance, even if the same PIA type is copied into multiple assemblies which are loaded together, they will not appear as different types to the executing program.

However there are some semantic differences. One is the independence of PIAs on the execution platform. this does lead to different execution semantics – typically in a good way.

A bigger difference is that COM signatures in linked types will be “dynamified”. This is described in the following.

## Dynamification of Linked COM members

COM methods are often designed for a language environment that is more dynamic than C#. This means that they will often return weakly typed results, and rely on the calling language to dynamically look up further operations on those results. More specifically such methods return results of the COM type variant, which can designate any object.

In PIAs we map these results – and also parameters of the same type – into the .NET type object. This causes the need for very frequent casting of such returned values into more specific types to which further operations can be applied. In some situations this leads to unnecessarily clumsy code, all caused by a mismatch between the actual language environment of the call – C# – and the environment that the method expects.

Now that C# has the dynamic type, it would seem more appropriate to map COM’s variant type into that. Unfortunately, out of compatibility concerns, we cannot change how PIAs represent the variant type, nor can we change how C# imports the PIA types that are referenced in the existing manner.

However, because linked PIAs are a new feature, we can give different semantics to these, and indeed we do.

Specifically any COM method in a linked assembly which has the return type object will be treated in C# as if the return type is dynamic. Therefore, further calls on the result of such a method will be bound dynamically as described in §2.

## The COM runtime binder

Even though they are not dynamic objects, dynamic operations on COM objects are not dispatched by the C# runtime binder but instead by a special COM runtime binder which is shared among multiple languages. This means that features such as default properties and indexed properties will be respected, even though such features are unknown to C#.

## 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

excel.Cells[1, 1].Value = "Process Name"; // dynamic property set  
 excel.Cells[1, 2].Value = "Memory Usage"; // dynamic property set

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; // dynamic property set  
 excel.Cells[i, 2].Value = p.WorkingSet; // dynamic property set  
 i++;  
 }

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

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.

# Co- and Contravariance

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; // error

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 problem 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;

There is no way we can put the wrong kind of thing into strings through objects, because objects doesn’t have any method that takes an element in as an argument. Co- and contravariance is about allowing assignments such as this in cases where it is safe.

## Covariance

In .NET 4.0 the IEnumerable<T> and IEnumerator<T>interfaces 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 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 IEnumerable<A> is implicitly reference convertible to IEnumerable<B> if A has an implicit reference conversion to B.

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

This is useful 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 some cumbersome wrapping would have had to be applied 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 result is that an IComparer<object> can in fact be considered an IComparer<string>. This may be surprising at first, but in fact makes perfect sense: If a comparer can compare *any* two objects, it can certainly also compare two strings. The interface is said to be “contravariant”.

A generic type can have both in and out modifiers on its type parameters, as is the case with the Func<…> delegate types in .NET 4.0:

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

Co- and contravariant type parameters can only be declared on interfaces and delegate types. Co- and contravariance only applies when there is a *reference* (or identity) 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.

## Syntax

Variance annotations on type parameters are only allowed on interface and delegate type declarations.

variant-type-parameter-list:  
< variant-type-parameters >

variant-type-parameters:  
attributesopt variance-annotationopt  type-parameter  
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:

in

out

The only difference from ordinary type-parameter-lists is the optional variance-annotation on each type parameter. If the variance annotation is out, the type parameter is said to be covariant. If the variance annotation is in, the type parameter is said to be contravariant. If there is no variance annotation, the type parameter is said to be invariant.

In the example

interface C<out X, in Y, Z>   
{  
 X M(Y y);

Z P { get; set; }  
}

X is covariant, Y is contravariant and Z is invariant.

## Variance safety

The occurrence of variance annotations in the type parameter list of a type restricts the places where types can occur within the type declaration.

A type T is output-unsafe if one of the following holds:

* T is a contravariant type parameter
* T is an array type with an output-unsafe element type
* T is an interface or delegate type S<A1,… AK> constructed from a generic type S<X1, .. XK> where for at least one Ai one of the following holds:
* Xi is covariant or invariant and Ai is output-unsafe.
* Xi is contravariant or invariant and Ai is input-safe.

A type T is input-unsafe if one of the following holds:

* T is a covariant type parameter
* T is an array type with an input-unsafe element type
* T is an interface or delegate type S<A1,… AK> constructed from a generic type S<X1, .. XK> where for at least one Ai one of the following holds:
* Xi is covariant or invariant and Ai is input-unsafe.
* Xi is contravariant or invariant and Ai is output-unsafe.

Intuitively, an output-unsafe type cannot appear in an output position, and an input-unsafe type cannot appear in an input position.

A type is output-safe if it is not output-unsafe. A type is input-safe if it is not input-unsafe.

## Interface declarations

The declaration syntax for delegate types is extended as follows:

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface identifier variant-type-parameter-listopt  
 interface-baseopt type-parameter-constraints-clausesopt interface-body ;opt

Every base interface of an interface must be output-safe.

### Interface methods

The return type must be either void or output-safe.

Each formal parameter type must be input-safe.

Each class type constraint, interface type constraint and type parameter constraint on any type parameter of the method must be input-safe.

These three rules ensure that any covariant or contravariant usage of the interface remains typesafe. For example,

interface I<out T> { void M<U>() where U : T; }

is illegal because the usage of T as a type parameter constraint on U is not input-safe.

Were this restriction not in place it would be possible to violate type safety in the following manner:

class B {}  
class D : B{}  
class E : B {}  
class C : I<D> { public void M<U>() {…} }  
…  
I<B> b = new C();  
b.M<E>();

This is actually a call to C.M<E>. But that call requires that E derive from D, so type safety would be violated here.

### Other interface members

The type of an interface property must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

The type of an interface event must be input-safe.

The formal parameter types of an interface indexer must be input-safe . Any out or ref formal parameter types must also be output-safe. Note that even out parameters are required to be input-safe. This is a limitiation of the underlying execution platform.

The type of an interface indexer must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

## Delegate declarations

The declaration syntax for delegate types is extended as follows:

delegate-declaration:  
attributesopt delegate-modifiersopt delegate return-type identifier variant-type-parameter-listopt   
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

The return type of a delegate must be either void, or output-safe.

The formal parameter types of a delegate must be input-safe. Any out or ref parameter types must additionally be output-safe. Note that even out parameters are required to be input-safe. This is a limitiation of the underlying execution platform.

## Conversions

A type T<A1, …, An> is variance-convertible to a type T<B1, …, Bn> if T is either an interface or a delegate type declared with the variant type parameters T<X1, …, Xn>, and for each variant type parameter Xi one of the following holds:

* Xi is covariant and an implicit reference or identity conversion exists from Ai to Bi
* Xi is contravariant and an implicit reference or identity conversion exists from Bi to Ai
* Xi is invariant and an identity conversion exists from Ai to Bi

### Implicit conversions

A reference type S has an implicit reference conversion to an interface or delegate type T if it has an implicit reference conversion to an interface or delegate type T0 and T0 is variance-convertible to T.

A value type S has an implicit boxing conversion to an interface type I if it has an implicit boxing conversion to an interface or delegate type I0 and I0 is variance-convertible to I.

A type parameter T has an implicit conversion to an interface type I if it has an implicit conversion to an interface or delegate type I0 and I0 is variance-convertible to I. At run-time, if T is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

### Explicit conversions

A reference type S has an explicit reference conversion to an interface or delegate type T if it has an explicit reference conversion to an interface or delegate type T0 and either T0 is variance-convertible to T or T is variance-convertible to T0.

A value type S has an explicit unboxing conversion from an interface type I if it has an explicit unboxing conversion from an interface or delegate type I0 and either I0 is variance-convertible to I or I is variance-convertible to I0.

## Type Inference

The type inference algorithm of §7.4 needs to be augmented in several ways to accommodate co- and contravariance. The most significant change is that the notion of bounds collected throughout the inference process is being refined into three different kinds of bounds: upper bounds, lower bounds and exact bounds. Each step that infers bounds is augmented to specify which kind of bound is inferred, and the “fixing” process which selects an inferred type based on the bounds is augmented to take the kind of bound into account.

Also this augmentation means that the algorithm depends on whether the parameters xi of the candidate method is defined as a ref, out or value parameter.

### The first phase

Section §7.4.2.1 is modified as follows:

For each of the method arguments Ei:

* If Ei is an anonymous function, an explicit parameter type inference (§7.4.2.7) is made from Ei to Ti
* Otherwise, if Ei has a type U and xi is a value parameter then a *lower-bound inference* is made from U to Ti.
* Otherwise, if Ei has a type U and xi is a ref or out parameter then an exact inference is made from U to Ti.
* Otherwise, no inference is made for this argument.

### Exact inferences

Section §7.4.2.8 is modified as follows:

An exact inference from a type U to a type V is made as follows:

* If V is one of the unfixed Xi then U is added to the set of exact bounds for Xi.
* Otherwise, sets V1…Vk and U1…Uk are determined by checking if any of the following cases apply:
* V is an array type V1[…] and U is an array type U1[…] of the same rank
* V is the type V1? and U is the type U1?
* V is a constructed type C<V1…Vk> and U is a constructed type C<U1…Uk>

If any of these cases apply then an exact inference is made from each Ui to the corresponding Vi.

* Otherwise no inferences are made.

### Lower-bound inferences

Section §7.4.2.9 is modified as follows:

A lower-bound inference from a type U to a type V is made as follows:

* If V is one of the unfixed Xi then U is added to the set of lower bounds for Xi.
* Otherwise, sets U1…Uk and V1…Vk are determined by checking if any of the following cases apply:
* V is an array type V1[…]and U is an array type U1[…] (or a type parameter whose effective base type is U1[…]) of the same rank
* V is one of IEnumerable<V1>, ICollection<V1> or IList<V1> and U is a one-dimensional array type U1[](or a type parameter whose effective base type is U1[])
* V is the type V1? and U is the type U1?
* V is a constructed class, struct, interface or delegate type C<V1…Vk> and there is a unique type C<U1…Uk> such that U (or, if U is a type parameter, its effective base class or any member of its effective interface set) is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) C<U1…Uk>.

(The “uniqueness” restriction means that in the case interface C<T>{} class U: C<X>, C<Y>{}, then no inference is made when inferring from U to C<T> because U1 could be X or Y.)

If any of these cases apply then an inference is made from each Ui to the corresponding Vi as follows:

* If Ui is not known to be a reference type then an *exact inference* is made
* Otherwise, if U is an array type then a *lower-bound inference* is made
* Otherwise, if V is C<V1…Vk> then inference depends on the i-th type parameter of C:
* If it is covariant then a *lower-bound inference* is made.
* If it is contravariant then an *upper-bound inference* is made.
* If it is invariant then an *exact inference* is made.
* Otherwise, no inferences are made.

### Upper-bound inferences

This section is added after §7.4.2.9:

An upper-bound inference from a type U to a type V is made as follows:

* If V is one of the unfixed Xi then U is added to the set of upper bounds for Xi.
* Otherwise, sets V1…Vk and U1…Uk are determined by checking if any of the following cases apply:
* U is an array type U1[…]and V is an array type V1[…]of the same rank
* U is one of IEnumerable<Ue>, ICollection<Ue> or IList<Ue> and V is a one-dimensional array type Ve[]
* U is the type U1? and V is the type V1?
* U is constructed class, struct, interface or delegate type C<U1…Uk> and V is a class, struct, interface or delegate type which is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) a unique type C<V1…Vk>

(The “uniqueness” restriction means that if we have interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}, then no inference is made when inferring from C<U1> to V<Q>. Inferences are not made from U1 to either X<Q> or Y<Q>.)

If any of these cases apply then an inference is made from each Ui to the corresponding Vi as follows:

* If Ui is not known to be a reference type then an *exact inference* is made
* Otherwise, if V is an array type then an *upper-bound inference* is made
* Otherwise, if U is C<U1…Uk> then inference depends on the i-th type parameter of C:
* If it is covariant then an *upper-bound inference* is made.
* If it is contravariant then a *lower-bound inference* is made.
* If it is invariant then an *exact inference* is made.
* Otherwise, no inferences are made.

### Fixing

Section §7.4.2.10 is modified as follows:

An unfixed type parameter Xi with a set of bounds is fixed as follows:

* The set of *candidate types* Uj starts out as the set of all types in the set of bounds for Xi.
* We then examine each bound for Xi in turn: For each exact bound U of Xi all types Uj which are not identical to U are removed from the candidate set. For each lower bound U of Xi all types Uj to which there is *not* an implicit conversion from U are removed from the candidate set. For each upper bound U of Xi all types Uj from which there is *not* an implicit conversion to U are removed from the candidate set.
* If among the remaining candidate types Uj there is a unique type V from which there is an implicit conversion to all the other candidate types, then Xi is fixed to V.
* Otherwise, type inference fails.