Sites, Binders, and Dynamic Object Interop Spec

Alex Turner and Bill Chiles

1 Introduction

The Dynamic Language Runtime's (DLR) mission is to enable an ecosystem of dynamic languages on .NET and to support developing libraries that work well with dynamic language features. Dynamic languages have become very popular in the last several years, and static languages are adopting affordances for dynamic operations on objects. The DLR helps language implementers move languages to .NET and enhance existing languages with dynamic features. Library authors can engage easily in this space too.

The history of dynamic dispatch on Microsoft platforms stretches back to the advent of COM and the IDispatch interface. A COM object which chose to implement IDispatch could either have its methods and properties invoked statically by binding calls to its vtable entries, or dynamically by retrieving a list of available names and IDs, and then invoking the desired ID. Visual Basic and VBScript then provided a robust abstraction over this capability by letting you write direct method calls that would bind statically (early-binding) if the underlying object had a specific type or dynamically (late-binding) if its type was Object.

While Visual Basic retained this ability during the migration to the .NET Framework, late-binding on .NET has been limited to COM libraries that implement IDispatch and reflection over the static shape of .NET types. It's become necessary to provide a new mechanism for objects to specify their desired runtime binding logic that's shared by all .NET dynamic languages and which maintains as much of the performance of static binding as possible.

A few years ago, implementations of fully dynamic languages such as IronPython began targeting the .NET Framework, including the CLR as it presently exists. However, these languages each had to provide their own systems to implement dynamic calls, and needed to implement their own optimizations for common operations. Also, while these languages were designed to provide a great experience against their own objects as well as static .NET types, one dynamic language could not consume objects defined in another.

With the dynamic sites concept introduced by the DLR, we've solved both the performance and language interop problems with dynamic language implementations on the .NET Framework.

1.1 Performance

A major motivation behind the DLR has been to optimize the performance of dynamic operations without hard-coding specific fast-paths in advance.

For example, in the original implementation of IronPython before the DLR, there were vast quantities of code in associated helper classes that looked like this:

```
object Add(object x, object y) {
    if (x is int) return IntOps.Add((int)x, y);
    if (x is double) return DoubleOps.Add((double)x, y);
    // ... lots more special cases ...
    // Fall back to reflecting over operand types for op_Addition method
}
```

… with this static method in the IntOps class:

```
object Add(int x, object y) {
    if (y is int) return x + (int)y; // modulo overflow handling
    if (y is double) return (double)x + (double)y;
    // ... lots more special cases ...
}
```

This was unfortunate for two reasons. One problem was that it bloated the resulting IronPython assemblies with optimized fast-paths for thousands of cases (about 180kb in the DLL). The larger issue, however, was that this only optimized for classes were known when the fast-paths were fixed. Any classes with user-defined conversions end up falling back to the full search reflection case on each execution of "x + y".

By implementing an adaptive caching system that caches the implementations for the actual sets of types observed by a given dynamic call site, both the generally common cases and the cases specific to a given application can be fast.

1.2 Language Interop

Assuming you were willing to do such optimizations yourself, it would be possible to create a high-performing .NET dynamic language today without the DLR, as the DLR is simply an API that sits on top of the CLR and does not require any new built-in CLR features. This is, in fact, what the first versions of IronPython did, directly compiling Python code to IL that called run-time helper methods defined by IronPython. As the IronPython compiler knew about the semantics of standard .NET types, it could also provide a great experience targeting the BCL and existing C#/VB assemblies as well.

However, this story breaks down when you want an IronPython object to call into, for example, an IronRuby library and dynamically invoke functions on the objects it gets back. With no standard mechanism to understand how to perform dynamic operations on an external dynamic object, IronPython could only treat the object as a .NET object, accessing its statically-known API. Most likely any visible methods will be an implementation detail of IronPython rather than a representation of the user’s intent.

The DLR provides for this common currency when calling into objects defined in other dynamic languages by establishing a protocol for those implementing dynamic objects. As a language creator or library author who wants other DLR-aware languages to consume your objects dynamically, you must have the objects implement IDynamicMetaObjectProvider, offering up appropriate implementations of DynamicMetaObject when IDynamicMetaObjectProvider.GetMetaObject() is called.

2 Dynamic Call Sites

Dynamic expressions are those expressions a language defines that may only be fully bound at runtime, usually based on the runtime types of its operands or parameters. However, for most languages, the binding of such operations takes a non-trivial amount of time. If such an operation occurs in a loop, or inside a commonly invoked function, we wish to avoid calling into the language’s runtime binder every time this code path executes.

To avoid repeatedly binding the same expressions, the DLR provides a caching mechanism, the **dynamic call site**. A call site encapsulates a language's binder and the dynamic expression or operation to perform. A language compiler for a regular .NET language, such as C# or Visual Basic, emits a call site where dynamic expressions occur. Dynamic languages built completely on the DLR, such as IronPython, use DynamicExpression nodes from Expression Trees v2 (which the DLR ships in .NET 4.0, which the expression compiler resolves to dynamic call sites.

Each dynamic call site keeps track how to perform the operation it represents. It learns different ways to perform the operation depending on the data (operands) that flows into the site. The binder provides these implementations of the operation along with restrictions as to when the site can correctly use the implementations. We refer to these restrictions and implementations together as rules. The call site compiles the rules it sees into a target delegate that embodies the site' cache.

2.1 Rules

To explore rules, let’s consider the case of adding two dynamic operands in a DLR-aware language:

```
d1 + d2
```

Let’s assume that the first time we hit this expression, both operands have a runtime type of int. The operations’ dynamic call site has not yet bound any implementations for this addition site, and so it will immediately ask the runtime binder what to do. In this case, the runtime binder examines the types of the operands and decides to ensure that each operand is an int and then performs addition. It therefore returns an expression tree representing this:

```
(int)d1 + (int)d2
```

This expression tree serves as the **implementation**, which is the specific meaning the runtime binder has given to this dynamic expression in this exact situation. This expression tree alone would be sufficient to finish evaluating this expression, as it could be wrapped in a lambda expression, compiled into a delegate, and executed on the given operands. However, to implement a cache we also need to understand the extent of the situations in which we can reuse this implementation in the future. Is this implementation applicable whenever both operands are typed as int, or must the operands have not just the type int but perhaps an exact values encountered by this expression? The conditions under which an implementation applies in the future are known as the **test**.

For example, in the case above, the test returned by the binder might say that this implementation applies whenever the types of both arguments are int. This can be thought of as an if-condition that wraps the implementation:

```
if (d1 is int && d2 is int) {
    return (int)d1 + (int)d2;
}
```

By providing this implementation and this test, the binder is saying that in any case where d1 and d2 both have the runtime type int, the correct implementation will always be to cast d1 and d2 to int and add the results. This combination of a test, plus an implementation that is applicable whenever that test is met is what forms a **rule**. These rules are represented in the DLR as compound expressions formed from the implementation expression and the test.

A rule may also be generated for a failed binding to optimize future calls where it's guaranteed to fail again. For example, if the left operand was of type MyClass and this class does not have a user-defined + operator, the following rule may be returned:

```
if (d1 != null && d1.GetType() == typeof(MyClass)) {
      throw new InvalidOperationException("Runtime binding failed");
}
```

This rule chooses to test type identity so that it's only applicable when the type is *exactly* MyClass, as an instance of a derived class may derive its own + operator. If the language was such that it could be known that no derived class is allowed to define a + operator, or perhaps that this type is sealed, the binder could return a more efficient is-test. This is the type of decision the language binder makes as it is the arbiter of its language's semantics.

Languages with highly dynamic objects may define more advanced tests as well. For example, if an object allows methods to be added or removed at runtime, it's not correct to simply cache what to do when that method is invoked for some operand types, as the method may be different or missing on the next invocation. In this case, the object may choose to keep track of a version number, incrementing the version each time a method is added or removed from the object. The test would then check both that the parameter types match, and that the version number is still the same. In the case that the version number has changed, the test would fail, and this rule would no longer apply. The site would then call the binder to bind again.

2.2 CallSiteBinder

The component of a dynamic site that actually accepts an operation's arguments at runtime and produces rules is known as a **binder**. A binder encapsulates the runtime semantics of the language that emitted the call site as well as any information the language needs at run time to bind the operation. Each dynamic call site the compiler emits has an instance of a binder. There are several kinds of binders for different kinds of operations, each deriving from CallSiteBinder. At runtime, this class will perform the binding the compiler is skipping during compile-time.

To actually perform binding and produce a rule, the CallSiteBinder's BindDelegate method is called. BindDelegate produces a rule delegate for the operation encoded by this binder by passing the parameters provided to the abstract Bind method to produce an expression tree representing the bound operation. This Bind method is implemented by the various subclasses of CallSiteBinder, including the primary set of DynamicMetaObject-based subclasses which allow for interop between objects of various languages.

An instance of a binder should encode all statically available information that distinguishes this operation from a similar operation. For our addition example, the language may define a general OperationBinder class, and then specify when constructing an instance of OperationBinder that the site's specific operation is addition.

It is up to a language to decide the specific static information it will need at runtime in addition to the list of arguments to bind this operation accurately. For a method invocation, say `d.Foo(bar, 123)`, the binder instance encodes the method name Foo, but may also choose to encode facts like the following if they are material to binding this invocation at runtime:

- the second parameter was passed a literal value
- named argument-passing was not used

It is entirely up to each language what compile-time information it chooses to encode in its binder for a given operation. Languages should be sure to reuse canonical binder instances, however, taking care to avoid generating two binders of the same type with the same compile-time information. As described below in the L2 cache section, equivalent call sites that share a common binder instance may share rules and avoid rebinding operations already bound elsewhere in the program.

2.3 CallSite<T>

When a compiler emits a dynamic call site, it must first generate a **CallSite<T>** object by calling the static method CallSite<T>.Create. The compiler passes in the language-specific binder it wants this call site to use to bind operations at runtime. The T in the CallSite<T> is the delegate type that provides the signature for the site's target delegate that holds the compiled rule cache. T's delegate type often just takes Object as the type of each of its arguments and its return value, as a call site may encounter or return various types. However, further optimization is possible in more restricted situations by using more specific types. For example, in the expression `a > 3`, a compiler can encode in the delegate signature that the right argument is fixed as int, and the return type is fixed as bool, for any possible value of a.

To allow for more advanced caching behavior at a given dynamic call site, as well as caching of rules across similar dynamic call sites, there are three distinct **caching levels** used, known as the L0, L1, and L2 caches. The code emitted at a dynamic site searches the L0 cache by invoking the site's Target delegate, which will fall back to the L1 and L2 caches upon a cache miss. Only if all three caches miss will the site call the runtime binder to bind the operation.

2.3.1 L0 Cache: CallSite's Target Delegate

The core of the caching system is the **Target** delegate on each dynamic site, also called the site's **L0 cache**, which points to a compiled dynamic method. This method implements the dynamic site's current caching strategy by baking some subset of the rules the dynamic site has seen into a single pre-compiled method. Each dynamic site also contains an **Update** delegate which is responsible for handling L0 cache misses by continuing on to search the L1 and L2 caches, eventually falling back to the runtime binder's BindDelegate method to produce a new rule. The current method referenced by the Target delegate always contains a call to Update in case the rule does not match.

For example, an L0 Target method for a call site that has most recently adding ints might look like this:

```
if (d1 is int && d2 is int) {
    return (int)d1 + (int)d2;
}
return site.Update(site, d1, d2);
```

Note that the CLR's JIT will compile this down to an extremely tight method (for example, the `is` tests should be just a few machine instructions, and the casts will disappear). This means that while it's somewhat expensive to update the L0 cache for a new set of encountered types (as this requires compiling a new dynamic method), for L0 cache hits execution will come very close to the performance of static code.

2.3.2 L1 Cache: CallSite's Rule Set

As it takes time to generate the dynamic methods needed for the L0 cache, we don't want to throw them away when a new set of types is encountered. For this reason, a call site will keep track of the 10 most recently seen dynamic methods that it has generated in its **L1 cache**. When there is an L0 cache miss, and the Update method is called, each dynamic method in the L1 cache is called in turn until there's a cache hit. When a hit is found, the L0 cache is updated to point to this method once again. If not cache hit occurs, the call site proceeds to the L2 cache (see below).

2.3.3 L2 Cache: Combined Rule Sets of All Equivalent CallSites

Each dynamic call site has a delegate type and a binder instance. For each unique binder instance, a separate **L2 cache** is maintained that allows the sharing of rules between all call sites that reference this binder instance. The L2 cache helps eliminate startup costs for call sites that bind operations equivalent to those already bound elsewhere. Since the L2 cache may hold 100 delegates (compared to 10 in the L1 cache), it also provides an extra mechanism to improve the performance of highly polymorphic sites that encounter many various types.

Upon an L1 cache miss the binder would otherwise be required to generate a new dynamic method, either because this particular call site has not yet bound that operation, or because it has fallen off the end of this site's L1 cache. The shared L2 cache solves both of these problems by providing a larger cache that retains the rules seen by all equivalent call sites. An L2 hit means that the rule can be added back to the site's L1 cache and L0 target delegate without regenerating the dynamic method.

To facilitate this sharing of rules between call sites, language compilers and others who produce call sites must be sure to reuse a canonical binder instance across all "equivalent" call sites. It is up to a given compiler what set of criteria is used to determine call site equivalence, but the set must include all the site metadata that could influence rule production. For example, in C#, an addition operation may appear in either a checked or unchecked arithmetic context, which determines overflow behavior. As this context affects the semantics of arithmetic operations bound in C#, its compiler would only share a binder instance between "Add" call sites if they are all "checked" or all "unchecked". This way, an Add rule generated in a checked context would not be shared in an unchecked context.

2.3.4 Other Optimizations

The general architecture of the DLR's call sites and rules is open to several optimization techniques. For example, the rule expression generated for a given execution of a call site may follow a similar template as the last rule generated, differing only in some ConstantExpression nodes. The DLR can recognize rules that are similar in this way and combine them into one compiled rule that replaces each differing ConstantExpression with a ParameterExpression that expects the value as a parameter. This saves the cost of rebuilding and re-JIT'ing the Target delegate as the constant changes. This is one example of various techniques call sites allow us to employ in the DLR.

3 IDynamicMetaObjectProvider and DynamicMetaObject

With just the dynamic call sites and binders logic defined above, you can already imagine implementing a language which takes advantage of the DLR's caching system. You could emit a dynamic call site for each dynamic operation and write a binder that produces rules based on the types of the operands provided. Your binder could understand dynamic dispatch to static .NET types (whose static structure can be gleaned from reflection), as well as dispatch to its own dynamic types, which it natively understands.

However, what if you want the binder for your language to dispatch to functions on objects created by another dynamic language? In this case, reflection will not help you as it would only show you the static implementation details behind that type's dynamic façade. Even if the object were to offer a list of possible operations to perform, there are still two problems. You would have to rebuild your language every time some new language or object showed up with a new operation your language didn't know about. You would also really like to perform these operations with the semantics of the target object's language, not your language, to ensure that object models relying on quirks of that language's semantics still work as designed.

Also, a library author might have a very dynamic problem domain they want to model. For example, if you have a library that supports drilling into XML data, you might prefer that your users' code to look like Customers.Address.ZipCode rather than XElement.Element("address").Element ("zipcode"), with the library allowing dynamic access to its data. Another example would be a library doing something similar with JSON objects returned from web services, or other highly dynamic data sources.

What's needed is a mechanism through which objects defined in various dynamic languages can offer to *bind their own operations* in the way that they see fit, while still obtaining the benefits of the DLR's caching infrastructure. The most visible part of this mechanism is the IDynamicMetaObjectProvider interface, which classes implement to offer dynamic dispatch support to a consuming language. The DynamicMetaObject class complements IDynamicMetaObjectProvider and serves as the common representation of dynamic metadata, allowing operations to be dispatched on dynamic objects.

3.1 IDynamicMetaObjectProvider

An object that offers up either its source language's semantics or its own type's custom dispatch semantics during dynamic dispatch must implement **IDynamicMetaObjectProvider**. The IDynamicMetaObjectProvider interface has a single method, GetMetaObject, which returns a DynamicMetaObject that represents this specific object's binding logic. A key strategy for DLR languages is to use .NET's Object as the root of their type hierarchy, and to use regular .NET objects when possible (such as IronPython using .NET strings). However, the IDynamicMetaObjectProvider protocol may still be needed for these base types at times, such as with IronRuby's mutable strings.

Library authors (even those using static languages) might implement IDynamicMetaObjectProvider so that their objects can present a dynamic façade in addition to their static interface. These objects can then present a better programming experience to dynamic languages as well as enable lighterweight syntax in languages such as C# with the 'dynamic' keyword.

3.2 DynamicMetaObject

An instance of **DynamicMetaObject** represents the binding logic for a given object, as well as the result for a given expression that's been bound. It has methods that allow you to continue composing operations to bind more complex expressions.

The three major components of a DynamicMetaObject are:

- The value, the underlying object or the result of an expression if it has one, along with information on its type.
- An expression tree, which represents the result of binding thus far.

- A set of restrictions, which represent the requirements gathered along the way for when this expression tree can serve as a valid implementation.

The expression tree and set of restrictions should feel familiar as they are similar in purpose to the implementation and test within a rule. In fact, when a DynamicMetaObject-aware binder such as DynamicMetaObjectBinder is asked to bind a given DynamicMetaObject, it uses the expression tree as the implementation and transforms the restrictions into the test for the rule.

Restrictions should be mostly of a static or invariant nature. For example, they test whether an object is of a specific static type or is a specific instance, something that will not change due to side effects during evaluation of this expression. The expression tree produced by the DynamicMetaObject binding may supply other tests that vary on dynamic state. For example, for objects whose member list itself is mutable, the expression tree may contain a test of an exact version number of the type. If this dynamic version test fails, the rule may be outdated, and the call site needs to call the binder and update the cache.

DynamicMetaObjects may also be composed to support merging several operations or compound dynamic expressions into a single call site. For example, `a.b.c.d()` would otherwise compile to three or four distinct dynamic call sites, calling three or four Target delegates each time the operation is invoked. Instead, the compiler may generate a single call site that represents the compound operation. Then, for each pass through this call site at runtime, the binder can decide how much of this compound expression it can bind in advance, knowing just the type of the source, `a`. If `a` has a certain static type, the return type of `a.b` may then be known, as well as `a.b.c` and the invocation `a.b.c.d()`, even before the methods are executed. In this case, the binder can return a single rule whose implementation covers the entire invocation.

3.3 DynamicMetaObjectBinder

If a language wants to participate not just in the DLR’s caching system, but also interop fully with other dynamic languages, its binders need to derive from one of the subclasses of **DynamicMetaObjectBinder**.

DynamicMetaObjectBinder acts as a standard interoperability binder. Its subclasses represent various standard operations shared between most languages, such as GetMemberBinder and InvokeMemberBinder. These binders encode the standard static information expected across languages by these operations (such as the name of a method to invoke for InvokeMemberBinder). At runtime, DynamicMetaObject’s Bind method accepts a target DynamicMetaObject and an array of argument objects. This Bind method, however, does not perform the binding itself. Instead, it first generates a DynamicMetaObject for each operand or argument as follows:

- **For objects that implement IDynamicMetaObjectProvider:** The Bind method calls .GetMetaObject() to get a custom DynamicMetaObject.
- **For all other static .NET objects:** The DLR generates a simple DynamicMetaObject that falls back to the semantics of the language, as represented by this binder class.

This subclass of DynamicMetaObjectBinder then calls out to the relevant Bind… method on the target DynamicMetaObject (such as BindInvokeMember), passing it the argument DynamicMetaObjects. By convention all binders defer to the DynamicMetaObjects before imposing their language's binding logic on the operation. In many cases, the

DynamicMetaObject is owned by the language, or if it is a default DLR .NET DynamicMetaObject, it falls back to the language binder for deciding how to bind to static .NET objects. Deferring to the DynamicMetaObject first is important for interoperability.

Binding using DynamicMetaObjectBinders takes place at a higher level by using DynamicMetaObjects instead of the .NET objects themselves. This lets the designer of dynamic argument objects or operands retain control of how their operations are dispatched. Each object's own defined semantics always take highest precedence, regardless of the language in which the object is used, and what other objects are used alongside them. Also, because all languages derive from the same common binder classes, the L0 target methods that are generated can cache implementations across the various dynamic languages and libraries where objects are defined, enabling high performance with semantics that are true to the source.

3.3.1 Fallback Methods – Implementing the Language's Semantics

A major principle in the DLR design is that "the object is king". This is why DynamicMetaObjectBinder always delegates binding first to the target's DynamicMetaObject, which can dispatch the operation with the semantics it desires, often those of the source language that object was defined in. This helps make interacting with dynamic object models feel as natural from other languages as it is from the language the object model was designed for.

However, there are often times when you'll write code that uses a language feature available in your language, but not in the language of your target object. For example, let's say you're coding in Python, which provides an implicit member __class__ on all objects that returns the object's type. When dynamically dispatching member accesses in Python, you'll want the __class__ member to be available not just on Python objects, but also on standard .NET objects, as well as objects defined in other dynamic languages. This is where Fallback methods come in.

Each of the DynamicMetaObjectBinder subclasses defines a **fallback method** for the specific operation it represents. This fallback method implements the language's own semantics for the given operation, to be used when the object is unable to bind such an operation itself. While a language's fallback methods may supply whatever semantics the language chooses, the intention is that the language exposes a reasonable approximation of its compile-time semantics at runtime, applied to the actual runtime types of the objects encountered.

For example, the SetMemberBinder class defines an abstract FallbackSetMember method that languages override to implement their dynamic member assignment semantics. At a member assignment call site, if the target DynamicMetaObject's BindSetMember method can't dispatch the assignment itself, it can delegate back to the language's SetMemberBinder. The DynamicMetaObject does this by calling the binder's FallbackSetMember method, passing the target DynamicMetaObject (itself) and the value to assign to the member.

There are specific subclasses of DynamicMetaObjectBinder for each of 12 common language features in the interoperability protocol. For each kind of operation for which you support binding in your language, you must implement a binder that derives from its class. The API Reference section below describes these operation binder abstract classes and provides further detail on implementing them.

3.3.2 Error Suggestions

An MO that wants even finer control over the “fallback dance” may also choose to pass an **error suggestion** to a language’s fallback method. An error suggestion lets a dynamic object supply a binding recommendation to the language binder, which the language binder should use if it fails its own binding process.

Many complex binding behaviors may be constructed in this way, but a typical technique is to facilitate a “static-first” binding strategy. A dynamic object that wants to prefer static members over dynamic members of the same name will immediately call the language binder’s Fallback method when asked to bind an operation. The dynamic object will supply an error suggestion MO to the language binder that contains its logic on how to perform dynamic member dispatch. The language will then attempt its own binding to the object’s static members. Only if this binding fails, will the language resort to the object’s error suggestion, which will then attempt dynamic binding.

This strategy of letting the language bind first prevents a dynamic member from shadowing a known static member available on an object, and is used by the DynamicObject abstract class described later in this document.

3.4 Dynamic Binding Walkthroughs

The following sections walk through some simple binding operations to help demonstrate the dynamic binding “dance” that occurs between the target object’s MetaObject and the source language’s DynamicMetaObjectBinder. These sections also demonstrate the caching mechanism that works with the binding process to accelerate a given operation the second time it’s encountered.

3.4.1 a + b

This walkthrough describes how the C# Compiler will dispatch the operation a + b dynamically:

```
        c = a + b;
```

The C# language specifies that operations are dispatched dynamically when at least one operand has the static C# type, *dynamic*, introduced in C# 4.0. In this case, we’ll assume that at least the left operand has the type *dynamic*.

The following is the equivalent C# source code for what the C# compiler will emit for the dynamic call site above:

```
if (SiteContainer.Site1 == null)
{
    SiteContainer.Site1 =
        CallSite<Func<CallSite, object, object, object>>.Create(
            CSharpBinaryOperationBinderFactory.Create(
                ExpressionType.Add, false, false,
                new CSharpArgumentInfo[] {
                    new CSharpArgumentInfo(0, null),
                    new CSharpArgumentInfo(0, null)
                }
            )
        );
}
object c = SiteContainer.Site1.Target(SiteContainer.Site1, a, b);
```

The lifetime of the CallSite proceeds as follows:

1. The first block ensures that a singleton CallSite object for this call site has been initialized:
    a. First, we check that we haven't already generated this CallSite object. To avoid generating the CallSite object twice, C# stores it in a static field in a generated SiteContainer class. Each call site will get its own field.
    b. We then call CallSite<T>.Create to generate the CallSite object for this site. In this case, the delegate type T is the type Func<CallSite, object, object, object>. This means that the Target delegates generated by the DLR to cache and implement this operation will expect to take two objects as operands (along with the CallSite itself) and then return an object as the result. Taking two values and returning one is what we'd expect for a binary operation such as +.
    c. We use a factory method from the C# Runtime to generate an instance of C#'s specific BinaryOperationBinder subclass. The factory method will ensure that we get the canonical binder instance for this call site. The method's first parameter specifies that this BinaryOperation is an Add operation, while the remaining parameters supply C#-specific information, such as whether this operation occurs in a C# checked-arithmetic context and whether either operand was a literal value. Languages are free to stash away in their binder classes this sort of compile-time information if they'll need to bind accurately at runtime.
2. Once our CallSite is initialized, the Target delegate is invoked for the first time to bind and perform the operation as follows:
    a. The Target delegate (L0 cache) starts out as simply a copy of the Update delegate, which is called to bind the operation and then update the Target delegate. Invoking the Update delegate indicates an L0 cache miss, which is to be expected as this is the first time through this call site.
    b. Before binding, the L1 and L2 caches are checked to see if an applicable binding already exists for the operand types provided. As this is the first time we have hit this particular call site, there will be no match in the call-site-specific L1 cache. We'll assume for this walkthrough that there was no match in the global L2 cache as well.
    c. To actually perform the binding, the DynamicMetaObjectBinder base class first produces a DynamicMetaObject (MO) for each operand using DynamicMetaObject.Create. The MO produced for each operand depends on that operand's type:
        - For dynamic objects (objects that implement IDynamicMetaObjectProvider), an MO is produced by calling the object's GetMetaObject method. This allows any custom binding semantics the object defines to take precedence over C# semantics by default, as the 12 standard interoperability binders that derive from

DyanmicMetaObjectBinder will always defer to an object's MO before the source language.
  - For all other .NET objects, a default MO implementation is returned which will always "fall back" to the language binder (in this case, C#'s binder) to bind operations.

d. With our operands' MOs in hand, DynamicMetaObject.BindBinaryOperation is called on the left operand's MO, passing it the right MO as an argument. How this binding proceeds depends on whether the left MO defines its own semantics for +:

  - If the left MO is from a Python object, for example, it will implement Python's semantics for binding the + operator and produce an expression tree to that effect, based on the specific Python type of the left operand and the CLR or Python type of the right operand. This expression tree is returned within a new MO as the result of the operation, along with a new set of restrictions for when this tree is a valid binding. In this case, step e below is skipped.
  - However, not all MOs implement all dynamic operations. If this MO is from a dynamic library whose objects do not define custom semantics for +, or if this is the default .NET MO produced for non-dynamic objects, the BindBinaryOperation method will not generate an MO itself. Instead, it will "fall back" to the language binder passed in to BindBinaryOperation by calling FallbackBinaryOperation on our instance of CSharpBinaryOperationBinder.

e. If the left operand's MO can't bind + itself, the MO calls the C# binder's FallbackBinaryOperation method to perform the binding. This allows the C# binder to provide its own binding semantics for +, based on the runtime types and values of each operand, along with any other compile-time information it had stashed away. The resulting expression tree that represents the binding will be returned in a new MO with a new restrictions set, just as in the Python case.

f. The CallSiteBinder base now produces a delegate of type Func<CallSite, object, object, object> that implements the returned expression tree and the restrictions contained by the resulting MO. This delegate then replaces the Target delegate (L0 cache) for this call site, and is stored in the L1 and L2 caches to speed up future binding as well. Later, when this delegate is pulled from the cache, it will test these restrictions, executing the compiled expression if this binding applies to the operands provided, and calling the Update delegate for a new binding if it does not.

g. Now that we've updated the Target delegate, its compiled expression is executed to perform the operation on the operands provided.

3. Later, this same call site may be hit again. The stored CallSite object is retrieved and its Target delegate is invoked to perform the operation and any necessary binding, based on the results of the delegate's restriction test:

   - If we see the same operand types as we did previously, it's likely that the Target delegate's restriction test will succeed and the operation will be performed. In this case, the only overhead vs. a statically-bound operation is the delegate invocation and a type-test for each operand, both of which are quite fast in CLR 4.0.

- If we see different types, the Target delegate's restriction will fail and the Update delegate will be called again. In this case, if no match is found in the L1 or L2 caches, binding will proceed as it did before, with the binding result replacing the current contents of the Target delegate. This will leave the new binding in the L0 cache, with both the new and the previous bindings in the L1 and L2 caches.

3.4.2 a.Foo(b)

The process of binding other operations such as SetMember and GetIndex proceeds in much the same way, with any minor differences listed in the API sections below. One operation worth calling out specifically, however, is InvokeMember.

Languages may implement member invocations such as a.Foo(b) in one of two ways:

- A GetMember operation followed by an Invoke operation, as in Python. This allows the intermediate first-class function object to be accessed and manipulated directly.
- An atomic InvokeMember operation that both resolves the member access and dispatches the invocation, as in C# or Ruby.

To be compatible with all DLR-aware languages, DynamicMetaObject (MO) implementations that bind member invocations must support both Invoke and InvokeMember operations. For example, when C# is binding an InvokeMember operation against a dynamic object with first-class function members (like Python), the fallback mechanism is a bit more involved to allow Python to perform the GetMember and C# to perform the Invoke half. Giving control of Invoke back to C# allows it to control the implicit conversions of arguments, as well as other C#-specific binding behaviors.

Consider the C# Compiler dispatching the following member invocation dynamically, where a is a Python object and b is a standard .NET object:

```
c = a.Foo(b);
```

Binding proceeds mostly as described for a+b, with some important differences around fallback methods:

1. A CallSite singleton for this CSharpInvokeMemberBinder is initialized as normal.
2. The call site's Target delegate is then invoked, calling into the Update delegate that will bind the operation after failing to find a binding in the L1 or L2 caches.
    a. A Python MO is produced for target a, and a default .NET MO is produced for argument b.
    b. DynamicMetaObject.BindInvokeMember is called on the target's Python MO, passing it the argument MO. Python will then dispatch the GetMember half of this operation itself by doing its standard GetMember lookup on the member name to produce an intermediate MO for the first-class function object represented by that member. This MO contains an Expression representing a callable object, but the MO may not have a concrete value stored in it yet.

c. Python's BindInvokeMember implementation will then fall back to the C# binder to invoke this function object by calling InvokeMemberBinder.FallbackInvoke, and passing it the intermediate function object's MO. The C# binder will bind this function call as a standard delegate invocation, applying C#'s own argument conversion and error semantics.

d. A delegate is then produced as normal for the resulting MO, updating the call site's Target delegate and executing the compiled expression to perform the operation.

3. When this call site is hit again, the Target delegate is executed, checking the L1 and L2 cache and perhaps rebinding if necessary. If no dynamic member is found with the name "Foo", or if the MO is owned by an InvokeMember-oriented language such as Ruby, the MO may simply call FallbackInvokeMember rather than splitting lookup and invocation by calling FallbackInvoke.

4 DynamicObject and ExpandoObject

We make life much simpler for library authors who want to create objects in static languages so that the object can behave dynamically and participate in the interoperability and performance of dynamic call sites. Library authors can avoid the full power of DynamicMetaObjects, but they can also employ DynamicMetaObjects if they wish. The DLR provides two higher level abstractions over DynamicMetaObject: DynamicObject and ExpandoObject. For most APIs, these objects provide more than enough performance and flexibility to expose your functionality to dynamic hosts.

4.1 DynamicObject

The simplest way to give your own class custom dynamic dispatch semantics is to derive from the **DynamicObject** base class. DynamicObject lets your objects fully participate in the dynamic object interoperability protocol, supporting the full set of operations available to objects that provide their own custom DynamicMetaObjects. DynamicObject lets you choose which operations to implement, and allows you to implement them much more easily than a language implementer who uses DynamicMetaObject directly.

DynamicObject provides a set of 12 virtual methods, each corresponding to a Bind... method defined on DynamicMetaObject. These methods represent the dynamic operations others may perform on your objects:

```
public abstract class DynamicObject : IDynamicMetaObjectProvider
{
    public virtual bool TryGetMember(GetMemberBinder binder,
                                     out object result)
    public virtual bool TrySetMember(SetMemberBinder binder,
                                     object value)
    public virtual bool TryDeleteMember(DeleteMemberBinder
binder)

    public virtual bool TryConvert(ConvertBinder binder,
                                   out object result)
    public virtual bool TryUnaryOperation
```

```
        (UnaryOperationBinder binder, out object result)
    public virtual bool TryBinaryOperation
        (BinaryOperationBinder binder, object arg,
         out object result)

    public virtual bool TryInvoke
        (InvokeBinder binder, object[] args, out object result)
    public virtual bool TryInvokeMember
        (InvokeMemberBinder binder, object[] args,
         out object result)
    public virtual bool TryCreateInstance
        (CreateInstanceBinder binder, object[] args,
         out object result)
    public virtual bool TryGetIndex
        (GetIndexBinder binder, object[] args, out object result)
    public virtual bool TrySetIndex
        (SetIndexBinder binder, object[] indexes, object value)
    public virtual bool TryDeleteIndex
        (DeleteIndexBinder binder, object[] indexes)
```

You could have your own TryGetMember implementation look up “Foo” in a dictionary, crawl through a dynamic model like XML, make a web request for a value, or some other custom operation. To do so, you would override the TryGetMember method and just implement whatever custom action you want to expose through member evaluation syntax. You return true from the method to indicate that your implementation has handled this situation, and supply the value you want returned as the out parameter, result.

By default, the methods that you don’t override on DynamicObject fall back to the language binder to do binding, offering no special behavior themselves. For example, let’s say you have a class, MyClass, derived from DynamicObject that does *not* override TryGetMember. You also have an instance of MyClass in a variable, myObject, of type C# 'dynamic'. If you evaluate `myObject.`Foo, the evaluation falls back to C#’s runtime binder, which will simply look for a field or property named Foo (using .NET reflection) defined as a member of MyClass. If there is none, the C# binder will store a binding in the cache that will throw a runtime binder exception in this situation. A more real example is that you do override TryGetMember to look in your dictionary of dynamic members, and you return false if you have no such members. The DynamicMetaObject for MyClass produces a rule that first looks for static members on MyClass, then calls TryGetMember which may return false, and finally throws a language-specific exception when it finds no members.

In the full glory of the interoperability protocol, a dynamic object implements IDynamicMetaObjectProvider and returns a DynamicMetaObject to represent the dynamic view of the object at hand. The DynamicMetaObject looks a lot like DynamicObject, but its methods have to return Expression Trees that plug directly into the DLR's dynamic caching mechanisms. This gives you a great deal of power, and the ability to squeeze out some extra efficiency, while DynamicObject gives you nearly the same power in a form much simpler to consume. With DynamicObject, you simply override methods for the dynamic operations in which your dynamic object should participate. The DLR automatically creates a DynamicMetaObject for your DynamicObject. This DynamicMetaObject creates Expression Trees (for the DLR’s caching system) that simply call your overridden DynamicObject methods.

4.2 ExpandoObject

The **ExpandoObject** class is an efficient implementation of a dynamic property bag provided for you by the DLR. It allows you to dynamically retrieve and set its member values, adding new members per instance as needed at runtime. Because ExpandoObject implements the standard DLR interface IDynamicMetaObjectProvider, it is portable between DLR-aware languages. You can create an instance of an ExpandoObject in C#, give its members specific values and functions, and pass it on to an IronPython function, which can then evaluate and invoke its members as if it was a standard Python object. ExpandoObject is a useful library class when you need a reliable, plain-vanilla dynamic object.

4.3 Further Reading

If you'd like to learn more about DynamicObject and ExpandoObject, check out the accompanying Getting Started with the DLR for Library Authors on the DLR CodePlex site under Specs and Docs. That document covers DynamicObject and ExpandoObject at a deeper level and includes an example of using DynamicMetaObject to optimize a dynamic library.

5 Dynamic Object Conventions

As a language implementer or library author, you may use the mechanisms described above to expose whatever arbitrary binding semantics you wish, including semantics that differ greatly from those of the language consuming your objects. The fallback system exposed by the DynamicMetaObjectBinder subclasses enables your objects to blend their semantics naturally with those of the language consuming them.

In addition to the fallback system, there are also some established conventions for dynamic interop that allow consuming languages to learn about the state of your objects in a standard way. By following these conventions, you can expose concepts such as enumerability or disposability in a way that can change at runtime.

5.1 Enumerability

An object that represents a sequence of other objects is considered to be *enumerable*. Enumerable objects may then be enumerated over in a language such as C# that supports "foreach" blocks, yielding each element in its sequence in turn.

In the static .NET world, objects are known to be enumerable if they implement the IEnumerable or IEnumerable<T> interface. This interface promises that your object implements a GetEnumerator method that returns an IEnumerator object, which can be used to move through your sequence. If you're implementing an object library and know that a given class is always enumerable, the easiest way to specify enumerability is simply to implement IEnumerable<T> as normal.

However, you may be implementing library or language objects which are only sometimes enumerable, or which may become enumerable over time due to some dynamic operation on the object's type or itself. For example, in Python an object specifies that it is enumerable by defining an __iter__ method. Since this method may be added dynamically at runtime, the

IronPython compiler cannot know whether to implement the IEnumerable interface on the objects it produces.

For these dynamic situations, an object may instead specify enumerability by binding a dynamic conversion to the IEnumerable type, succeeding if the object is indeed enumerable. Languages such as C# will first attempt to convert dynamic objects being "foreached" to the IEnumerable interface using a Convert call site.

Note that if the IEnumerator object you return from GetEnumerator is itself disposable, this object should implement IDisposable statically.

5.2 Disposability

An object that has resources to release when the object is no longer required is considered to be *disposable*. Disposable objects offer a .Dispose method to call when the object is to be released, but more often the calling language will provide some sort of "using" block which will automatically call an object's Dispose method when the block completes.

Similar to enumerability, objects may statically specify that they are disposable by implementing the IDisposable interface, and this is the most straightforward approach if given classes in your object library always require disposing.

If your objects can become disposable at runtime, you may encode this in a similar fashion to enumerability, by successfully binding dynamic conversions to IDisposable. Languages such as C# using a dynamic object in a context that requires disposal will first attempt to dynamically convert the object to the IDisposable interface with a Convert call site.

6 API Reference

The following sections provide a detailed API reference for the types introduced in the conceptual overview above, with definitions for each type's most important members.

6.1 DynamicMetaObject Class

An instance of **DynamicMetaObject** (abbreviated MO) represents the binding logic for a given object. Binding an operation on an MO returns another MO with an Expression property representing the accumulated binding up to this point. The methods defined on MO allow you to continue composing these operations to bind more complex expressions.

6.1.1 Class Summary

```
public class DynamicMetaObject {
    public static readonly DynamicMetaObject[] EmptyMetaObjects;

    public DynamicMetaObject
        (Expression expression, BindingRestrictions restrictions,
         Object value);
    public DynamicMetaObject
        (Expression expression, BindingRestrictions restrictions);
    public Expression Expression { get; }
    public Boolean HasValue { get; }
    public Type LimitType { get; }
    public BindingRestrictions Restrictions { get; }
    public Type RuntimeType { get; }
    public Object Value { get; }

    public virtual DynamicMetaObject BindBinaryOperation
        (BinaryOperationBinder binder, DynamicMetaObject arg);
    public virtual DynamicMetaObject BindConvert(ConvertBinder binder);
    public virtual DynamicMetaObject BindCreateInstance
        (CreateInstanceBinder binder, DynamicMetaObject[] args);
    public virtual DynamicMetaObject BindDeleteIndex
        (DeleteIndexBinder binder, DynamicMetaObject[] indexes);
    public virtual DynamicMetaObject BindDeleteMember
        (DeleteMemberBinder binder);
    public virtual DynamicMetaObject BindGetIndex
        (GetIndexBinder binder, DynamicMetaObject[] indexes);
    public virtual DynamicMetaObject BindGetMember
        (GetMemberBinder binder);
    public virtual DynamicMetaObject BindInvoke
        (InvokeBinder binder, DynamicMetaObject[] args);
    public virtual DynamicMetaObject BindInvokeMember
        (InvokeMemberBinder binder, DynamicMetaObject[] args);
    public virtual DynamicMetaObject BindSetIndex
        (SetIndexBinder binder, DynamicMetaObject[] indexes,
         DynamicMetaObject value);
    public virtual DynamicMetaObject BindSetMember
        (SetMemberBinder binder, DynamicMetaObject value);
    public virtual DynamicMetaObject BindUnaryOperation
        (UnaryOperationBinder binder);

    public static DynamicMetaObject Create
        (Object value, Expression expression);

    public virtual IEnumerable<String> GetDynamicMemberNames();
```

6.1.2 Bind*Operation* Methods

There is a Bind*Operation* method defined for each of the dynamic operation types that may be dispatched on a DynamicMetaObject (MO). Each Bind method accepts a target binder representing the call site at which the operation is performed. It also takes any operands required to bind the operation, such as values, arguments, or indexes. Each Bind method returns a DynamicMetaObject containing the expression that represents the result of binding the operation and the restrictions under which the binding applies.

Note that all Bind methods besides the BindConvert method should return a DynamicMetaObject instance with a specified LimitType of Object. To produce a specific result type, compilers should emit an explicit ConvertBinder call site that wraps the resulting MO.

6.1.3 Expression Property

Returns an expression that represents the result of binding operations on a previous MO. An MO will always have an expression, even if it is a ParameterExpression to access some value in a program, since you can only create DynamicMetaObjects by passing an expression.

6.1.4 Restrictions Property

Returns a BindingRestrictions object, which represents the constraints gathered during the binding process. The constraints capture when this MO's Expression property can serve as a valid implementation of the bound operations.

6.1.5 Value and HasValue Properties

The Value property returns the concrete object represented by the DynamicMetaObject. The HasValue property returns a Boolean value as to whether the Value property is valid; this lets you disambiguate null for HasValue == False vs. a real null value represented by the MO.

The great majority of DynamicMetaObjects will have a value, either because they were directly produced from a dynamic object by calling its GetMetaObject method, or because they are the intermediate result of a dynamic operation where partial binding has produced a value. There are certain cases, however, where a DynamicMetaObject is produced without an underlying value.

An example is the intermediate result of a compound operation such as InvokeMember. A DynamicMetaObject that prefers to bind InvokeMember operations as a GetMember followed by a separate Invoke will likely bind the GetMember half of an InvokeMember operation and produce a new MO representing this intermediate result. The the original MO would use FallbackInvoke to have the language binder bind the Invoke operation on the result. In this case, you should not provide a value to the new intermediate result MetaObject produced by the GetMember operation, as this could allow the source language's binder to perform its own Invoke binding against the static .NET type used for the intermediate object. Instead, omit the value.

NOTE, binders must be careful to detect DynamicMetaObjects passed with HasValue== False and call the binder's Defer method to produce a nested CallSite. The nested CallSite in the case above would have an InvokeBinder, and the nesting allows the arguments to all get evaluated before flowing into the new CallSite. Then the InvokeBinder will see all MOs with actual values so that it can perform a proper binding. Without the Defer call, the binder and original DynamicMetaObject described in the previous paragraph will infinitely loop in the original CallSite. There is a concrete example of this explained in the sympl.doc file on www.codeplex.com/dlr (see the "Documents and Specs" link on the home page).

6.1.6 LimitType Property

Returns the most specific type this DynamicMetaObject is known to be. This will be the actual runtime type of the value, if available, or else the static type of the bound expression (its Expression.Type value).

6.1.7 Create Static Method

Returns a new DynamicMetaObject instance that allows you to bind further operations against a given object.

This method requires the actual object in question along with an Expression object that can represent this object in the newly bound tree. For example, when using this factory to create a DynamicMetaObject for a method call argument, you might pass the actual argument value being supplied, along with a ParameterExpression for a local in the calling context to represent the argument in any resulting bindings. Without such an expression, binding expressions would have to bake in the actual value, making the binding rule very specific and not very re-usable in the CallSite's caching.

6.1.8 GetDynamicMemberNames Method

Returns a list of member names that this DynamicMetaObject asserts could be accessed dynamically on this MO’s value or bound expression. This list is useful while debugging to allow an IDE to populate the list of members available on an object within debugger tool windows, or to allow an interactive prompt to display the list of members on an object.

6.2 IDynamicMetaObjectProvider Interface

This interface marks an object as a dynamic object that participates in the DLR's dynamic object interoperability protocol. A type which implements this interface is advertising the ability to offer up its own custom dynamic dispatch semantics in addition to a source language’s standard .NET binding. The IDynamicMetaObjectProvider interface has a single method, GetMetaObject, which returns a DynamicMetaObject representing the binding logic for either this specific object, the object's type, or its defining language.

For example, an object representing an XML node may implement IDynamicMetaObjectProvider to allow dynamic member access to its descendant nodes, while the type used to represent IronPython objects would implement the interface to expose Python’s dispatch semantics to whichever language is ultimately consuming the object.

6.2.1 Class Summary

```
public interface IDynamicMetaObjectProvider {
    DynamicMetaObject GetMetaObject(Expression parameter);
```

6.2.2 GetMetaObject Method

Returns a DynamicMetaObject that supports binding dynamic operations on the current object. Objects that implement IDynamicMetaObjectProvider will define their own subclass of DynamicMetaObject that implements the desired dynamic binding semantics for the object and return an instance of this subclass when GetMetaObject is called.

6.3 CallSiteBinder Abstract Class

CallSiteBinder is an abstract class from which the binder classes for each of a language’s specific operations are derived.

The code emitted by a language compiler will create an instance of a CallSiteBinder subclass for each dynamic call site. When this call site is reached, its Bind method will be called to produce an expression tree that represents the bound action required to execute this operation for the particular arguments specified.

Most language implementers will not directly derive from CallSiteBinder, but rather from DynamicMetaObjectBinder, which allows the use of DynamicMetaObject instances as a common currency during binding. The scenarios for directly implementing CallSiteBinder would be for binding and caching highly language-specific concepts, such as IronPython's global variable references or imports statements, which will not need to participate in interop, but would benefit from call site caching.

6.3.1 Class Summary

```
public abstract class CallSiteBinder {
    protected CallSiteBinder();

    public static LabelTarget UpdateLabel { get; }

    public abstract Expression Bind
         (Object[] args,
          ReadOnlyCollection<ParameterExpression> parameters,
          LabelTarget returnLabel);

    public virtual T BindDelegate<T>
         (CallSite<T> site,
          Object[] args);

    protected void CacheTarget<T>(T target);
```

6.3.2 Bind Method

Returns the bound expression produced by binding the operation represented by this call site for the arguments provided. BindDelegate calls the Bind method when a binding for a given operation is not present in the cache.

The args parameter contains the argument values passed in to this instance of the binder, while the parameters parameter contains a ParameterExpression instance for each parameter. These ParameterExpression instances may be used when referring to the parameter variables themselves in the resulting expression tree.

The returnLabel parameter provides a LabelTarget instance representing the point in the caller to continue from when this operation returns. This label should be passed to Expression.Return to generate the GotoExpression used within the bound expression to return its value.

6.3.3 BindDelegate Method

If you need more fine-grained control over the process by which bound expressions are transformed into delegates for execution, you may optionally override the BindDelegate method. Typically, it is only necessary to override the Bind method.

This method is called when there is a cache miss, and may decide for itself whether to call Bind to generate expression trees, as well as how and whether to call CacheTarget to add a new delegate to the cache.

For example, this enables you to implement a hardcoded fast-path for operations you know will be common so that you won't need to bind their expression trees and generate delegates at runtime. You may also use this to implement your own custom caching strategy if you wish to optimize for particular patterns of use you know your objects will often encounter.

6.3.4 CacheTarget Method

This method caches a given delegate representing a rule and must be called by overridden implementations of BindDelegate to add rules to the cache.

6.3.5 UpdateLabel Static Property

Returns the LabelTarget to use in a GotoExpression from within an expression representing the binding result. A cached expression may perform a Goto to this label if the rule must now fail the binding process and request that the CallSite bind the expression again.

This is often used within a bound expression in the 'else' branch of a version-check condition. If the expression's version check indicates the object has changed shape since binding last occurred (for example, adding or removing members), the cached binding may no longer apply, and so the binding process must occur again.

6.4 DynamicMetaObjectBinder Abstract Class

If a language wants to participate not just in the DLR's caching system, but also interop fully with other dynamic languages, its binders need to derive from DynamicMetaObjectBinder. DynamicMetaObjectBinder extends CallSiteBinder to use DynamicMetaObjects as a common currency, allowing the exchange of dynamic objects between languages.

The 12 predefined subclasses of DynamicMetaObjectBinder represent various standard operations shared between most languages, such as GetMemberBinder and BinaryOperationBinder. By implementing its operations using these standard binder classes, a language may consume DynamicMetaObjects provided by other languages.

For example, when IronPython binds a GetMember operation on an IronRuby object, its binder's base gets the Ruby object's DynamicMetaObject and calls BindGetMember on it. By passing an instance of IronPython's GetMemberBinder implementation to Ruby's BindGetMember method, the Ruby object may handle requests for Python-specific members it doesn't understand, such as __class__, by falling back to the IronPython binder.

For other operations outside the scope of these predefined subclasses, a language may define its own binder types that derive directly from DynamicMetaObjectBinder. A language which requires richer dispatch semantics for a standard operation may also choose to define its own direct DynamicMetaObjectBinder implementation for such operations, with this special binder reverting to an implementation of the standard subclass only when explicitly performing interop with objects from another language.

6.4.1 Fallback Methods

Each abstract subclass of DynamicMetaObjectBinder defines one or more abstract Fallback methods to be implemented by the language implementer. Each subclass will at least define a method named Fallback**Operation** where **Operation** is the operation represented by that subclass. These methods are called when a DynamicMetaObject is unable to bind the operation represented by the binder instance and wishes to "fall back" to the language to attempt its own binding. Binders should always call the Fallback**Operation** method in case of binding errors as the source language may allow a given operation that the object itself does not, or if not, may have language-specific error semantics that should be observed, such as returning a sentinel value.

Each Fallback method accepts the following parameters:

- The target object of the operation
- Any relevant operands to the operation such as values, arguments, or indexes
- An optional errorSuggestion DynamicMetaObject representing the object's suggestion for how to bind this operation if the language is also unable to produce a valid binding. This is usually null, or an expression for looking up a member dynamically should the binder fail to find a static member with the name.

Some subclasses also allow the object to "fall back" to a different operation than is represented by this binder. These cases are described in the specific sections below.

Also, be sure to check the HasValue property of each of the DynamicMetaObjects passed to your Fallback methods. Ensure that each parameter has a valid value before proceeding with binding. If not, you must use the binder's Defer method to emit a nested dynamic call site that will obtain values for these parameters at runtime before proceeding with the rest of the Fallback implementation.

6.4.2 Placing Canonical Binders on CallSites

To support proper L2 caching and sharing of rules between dynamic call sites, each site must share its binder instance with any other site representing an equivalent operation, as defined by the source language.

For example, C# supports two arithmetic contexts, checked and unchecked, which determine the semantics during integer overflow. As this distinction affects binding, C# should never unify binder instances for call sites which differ on the checked context.

A compiler may proactively enforce canonicalization by limiting the instantiation of its binder instances to factory methods which can ensure that duplicate binder instances are not produced for equivalent sites.

6.4.3 COM Support

This support has been moved to Codeplex only, in the Microsoft.Dynamic.dll. It is the COM binding C# uses, but it uses it privately now.

To support consuming COM objects dynamically, your language's binder classes must explicitly attempt COM binding, as seen below for GetMember operations:

```
DynamicMetaObject com;
if (System.Dynamic.ComBinder.TryBindGetMember
                                    (this, target, out com)) {
     return com;
}
```

For more information on the members of ComBinder, you may download the full DLR sources, including the COM binder at the DLR CodePlex site.

6.4.4 Class Summary

```
public abstract class DynamicMetaObjectBinder : CallSiteBinder {
    protected DynamicMetaObjectBinder();

    public virtual Type ReturnType { get; }

    public abstract DynamicMetaObject Bind
        (DynamicMetaObject target, DynamicMetaObject[] args);
    public DynamicMetaObject Defer(params DynamicMetaObject[] args);
    public DynamicMetaObject Defer
        (DynamicMetaObject target, params DynamicMetaObject[] args);
    public Expression GetUpdateExpression(Type type);
```

6.4.5 Bind Method

Returns a DynamicMetaObject that represents the result of binding this operation on the target DynamicMetaObject with the set of arguments provided. Each of the 12 standard interoperability binders defined by the DLR overrides this method with a sealed implementation that calls on the target MO.

6.4.6 Defer Method

Returns a DynamicMetaObject containing a DynamicExpression that defers a branch of the binding process until execution, once the value of the target and all arguments are known. This DynamicExpression will ultimately resolve to a nested CallSite. This allows the DyanmicMetaObject operand that has no value to get evaluated before flowing into the nested site. The nested site sees a brand new MO representing the value produced in the outer CallSite argument execution.

Note that a binder may defer either to itself or to another binder. For example, one common use of Defer is within a language's InvokeMemberBinder to implement FallbackInvokeMember. When its Bind call receives an IDO as the target object, FallbackInvokeMember may choose to defer binding to an instance of InvokeBinder which can bind later when the type of the retrieved member is known.

6.4.7 ReturnType Property

Returns a Type object representing the return type of the bound expressions this binder class will produce. The general principle for the binder base classes in the DLR is that binders produce bound expressions with either Void or Object values, except for ConvertBinders which produce expressions of a specific type. In this way, a language which requires a value of a specific type from a dynamic expression will introduce an explicit ConvertBinder call site for this final conversion if it wishes to dispatch the conversion dynamically. Requiring this convention avoids very complicated conversion logic (and frankly guess work) in dynamic objects that would need to produce values appropriate for any language that happens to own the call site.

You only need to override this property if you choose to derive your binders directly from DynamicMetaObjectBinder, as the built-in DynamicMetaObjectBinder subclasses will override this property for you.

6.4.8 GetUpdateExpression Method

Returns an Expression that when executed will cause the currently executing operation to be bound again.

Even after the binding process searches through cached rules and finds an applicable binding, it may turn out that something has changed dynamically about a given object. An example would be a dynamic object which may have members added and removed. If the object has changed since the previous binding was performed, that binding may no longer be applicable.

When a chosen binding expression discovers that it is no longer an applicable binding for the current operation, it can branch to the Expression returned by GetUpdateExpression. This expression will cause the binding process that chose the current rule to attempt binding again without considering this rule. Most likely, no other rules will apply and a new binding will be produced for this operation.

6.5 GetMemberBinder Abstract Class

Represents an access to an object's member that retrieves the value. In some languages the value may be a first-class function, such as a delegate, that closes over the instance, o, which can later be invoked.

Example: o.m

If the member doesn't exist, the binder may return an expression that creates a new member with some language-specific default value, returns a sentinel value like $Undefined, throws an exception, etc.

6.5.1 Class Summary

```
public abstract class GetMemberBinder : DynamicMetaObjectBinder {
    protected GetMemberBinder(String name, Boolean ignoreCase);

    public Boolean IgnoreCase { get; }
    public String Name { get; }

    public abstract DynamicMetaObject FallbackGetMember
        (DynamicMetaObject target, DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackGetMember
        (DynamicMetaObject target);
```

6.5.2 Name Property

Returns the name of the member to retrieve.

6.5.3 IgnoreCase Property

Returns a Boolean value as to whether the member name lookup should be case-insensitive.

6.6 SetMemberBinder Abstract Class

Represents an access to an object's member that assigns a value.

Example: o.m = 12

If the member doesn't exist, the binder may return an expression that creates a new member to hold this value, throws an exception, etc.

By convention, SetMemberBinder implementations should return rules whose result value is the value being assigned. This allows chaining of assignments in languages where assignment is an expression rather than a statement (e.g. a = b = c).

6.6.1 Class Summary

```
public abstract class SetMemberBinder : DynamicMetaObjectBinder {
    protected SetMemberBinder(String name, Boolean ignoreCase);

    public Boolean IgnoreCase { get; }
    public String Name { get; }

    public abstract DynamicMetaObject FallbackSetMember
         (DynamicMetaObject target, DynamicMetaObject value,
          DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackSetMember
         (DynamicMetaObject target, DynamicMetaObject value);
```

6.6.2 Name Property

Returns the name of the member to assign.

6.6.3 IgnoreCase Property

Returns a Boolean value as to whether the member name lookup should be case-insensitive.

6.7 DeleteMemberBinder Abstract Class

Represents an access to an object's member that deletes the member.

Example: delete o.m

This may not be supported on all objects.

6.7.1 Class Summary

```
public abstract class DeleteMemberBinder : DynamicMetaObjectBinder {
    protected DeleteMemberBinder(String name, Boolean ignoreCase);

    public Boolean IgnoreCase { get; }
    public String Name { get; }

    public abstract DynamicMetaObject FallbackDeleteMember
         (DynamicMetaObject target, DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackDeleteMember
         (DynamicMetaObject target);
```

6.7.2 Name Property

Returns the name of the member to delete.

6.7.3 IgnoreCase Property

Returns a Boolean value as to whether the member name lookup should be case-insensitive.

6.8 GetIndexBinder Abstract Class

Represents an access to an indexed element of an object that retrieves the value.

Example: o[10] *or* o[“key”]

If the element doesn't exist, the binder may return an expression that creates a new element with some language-specific default value, return a sentinel value like $Undefined, throw an exception, etc.

6.8.1 Class Summary

```
public abstract class GetIndexBinder : DynamicMetaObjectBinder {
    protected GetIndexBinder(CallInfo CallInfo);

    public CallInfo CallInfo { get; }

    public abstract DynamicMetaObject FallbackGetIndex
        (DynamicMetaObject target, DynamicMetaObject[] indexes,
         DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackGetIndex
        (DynamicMetaObject target, DynamicMetaObject[] indexes);
```

6.8.2 CallInfo Property

Returns a CallInfo instance providing more information about the indexes provided. Note that the CallInfo instance only represents information about the index arguments and not the receiver of the indexing operation.

6.9 SetIndexBinder Abstract Class

Represents an access to an indexed element of an object that assigns a value.

Example: o[10] = 12 *or* o[“key”] = value

If the member doesn't exist, the binder may return an expression that creates a new element to hold this value, throw an exception, etc.

By convention, SetIndexBinder implementations should return rules whose result value is the value being assigned. This allows chaining of assignments in languages where assignment is an expression rather than a statement (e.g. a[1] = b[2] = c).

6.9.1 Class Summary

```
public abstract class SetIndexBinder : DynamicMetaObjectBinder {
    protected SetIndexBinder(CallInfo CallInfo);

    public CallInfo CallInfo { get; }

    public abstract DynamicMetaObject FallbackSetIndex
```

```
         (DynamicMetaObject target, DynamicMetaObject[] indexes,
          DynamicMetaObject value, DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackSetIndex
         (DynamicMetaObject target, DynamicMetaObject[] indexes,
          DynamicMetaObject value);
```

6.9.2 CallInfo Property

Returns a CallInfo instance providing more information about the indexes provided. Note that the CallInfo instance only represents information about the index arguments and not the receiver of the indexing operation.

6.10 DeleteIndexBinder Abstract Class

Represents an access to an indexed element of an object that deletes the element.

Example: delete o.m[10] *or* delete o["key"]

This may not be supported on all indexable objects.

6.10.1 Class Summary

```
public abstract class DeleteIndexBinder : DynamicMetaObjectBinder {
    protected DeleteIndexBinder(CallInfo CallInfo);

    public CallInfo CallInfo { get; }

    public abstract DynamicMetaObject FallbackDeleteIndex
         (DynamicMetaObject target, DynamicMetaObject[] indexes,
          DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackDeleteIndex
         (DynamicMetaObject target, DynamicMetaObject[] indexes);
```

6.10.2 CallInfo Property

Returns a CallInfo instance providing more information about the indexes provided. Note that the CallInfo instance only represents information about the index arguments and not the receiver of the indexing operation.

6.11 InvokeBinder Abstract Class

Represents invocation of an invocable object, such as a delegate or first-class function object.

Example: a(3)

6.11.1 Class Summary

```
public abstract class InvokeBinder : DynamicMetaObjectBinder {
    protected InvokeBinder(CallInfo CallInfo);

    public CallInfo CallInfo { get; }

    public abstract DynamicMetaObject FallbackInvoke
         (DynamicMetaObject target, DynamicMetaObject[] args,
         DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackInvoke(DynamicMetaObject target,
         DynamicMetaObject[] args);
```

6.11.2 CallInfo Property

Returns a CallInfo instance providing more information about the arguments provided. Note that the CallInfo instance only represents information about the arguments and not the receiver of the invocation.

6.12 InvokeMemberBinder Abstract Class

Represents invocation of an invocable member on an object, such as a method.

Example: a.b(3)

If invoking a member is an atomic operation in a language, its compiler can choose to generate sites using InvokeMemberBinder instead of GetMemberBinder nested within InvokeBinder. For example, in C#, a.b may be a method group representing multiple overloads and would have no intermediate object representation for a GetMemberBinder to return.

6.12.1 Class Summary

```
public abstract class InvokeMemberBinder : DynamicMetaObjectBinder {
    protected InvokeMemberBinder
        (String name, Boolean ignoreCase, CallInfo CallInfo);

    public CallInfo CallInfo { get; }
    public Boolean IgnoreCase { get; }
    public String Name { get; }

    public abstract DynamicMetaObject FallbackInvoke(
        DynamicMetaObject target, DynamicMetaObject[] args,
         DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackInvokeMember(
        DynamicMetaObject target, DynamicMetaObject[] args);
    public abstract DynamicMetaObject FallbackInvokeMember
        (DynamicMetaObject target, DynamicMetaObject[] args,
         DynamicMetaObject errorSuggestion);
```

6.12.2 Name Property

Returns the name of the member to invoke.

6.12.3 IgnoreCase Property

Returns a Boolean value as to whether the member name lookup should be case-insensitive.

6.12.4 CallInfo Property

Returns a CallInfo instance providing more information about the arguments provided. Note that the CallInfo instance only represents information about the index arguments and not the receiver of the indexing operation.

6.12.5 FallbackInvoke Method

When emitting a callsite for the operation “a.foo()”, a language may choose to emit either a single InvokeMemberBinder or a GetMemberBinder then an InvokeBinder. To support dynamic objects that expect GetMember + Invoke in languages that create InvokeMemberBinders, an InvokeMemberBinder must provide a FallbackInvoke implementation.

The BindInvokeMember method for such an object will perform the GetMember itself and then call FallbackInvoke to ask the language to perform its invocation semantics on that member. The easiest way to implement this method is simply to create a DynamicExpression with an InvokeBinder instance and dispatch to a nested call site that will perform the Invoke on the intermediate object.

6.13 CreateInstanceBinder Abstract Class

Represents instantiation of an object with a set of constructor arguments. The object represents a type, prototype function, or other language construct that supports instantiation.

Example: new X(3, 4, 5)

6.13.1 Class Summary

```
public abstract class CreateInstanceBinder : DynamicMetaObjectBinder {
    protected CreateInstanceBinder(CallInfo CallInfo);

    public CallInfo CallInfo { get; }

    public abstract DynamicMetaObject FallbackCreateInstance
        (DynamicMetaObject target, DynamicMetaObject[] args,
         DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackCreateInstance
        (DynamicMetaObject target, DynamicMetaObject[] args);
```

6.13.2 CallInfo Property

Returns a CallInfo instance providing more information about the constructor arguments provided.

6.14 ConvertBinder Abstract Class

Represents a conversion of an object to a target type.

This conversion may be marked as being an implicit compiler-inferred conversion, or an explicit conversion specified by the developer.

Example: (TargetType)o

6.14.1 Class Summary

```
public abstract class ConvertBinder : DynamicMetaObjectBinder {
    protected ConvertBinder(Type type, Boolean explicit);

    public Boolean Explicit { get; }
    public Type Type { get; }

    public abstract DynamicMetaObject FallbackConvert
        (DynamicMetaObject target, DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackConvert(DynamicMetaObject target);
```

6.14.2 Type Property

Returns a Type instance representing the target type of the conversion operation.

6.14.3 Explicit Property

Returns true if the conversion was specified explicitly in the source code, and false if the conversion was introduced implicitly by the compiler.

6.15 UnaryOperationBinder Abstract Class

Represents a unary operation.

Examples: -a, !a

Contains an ExpressionType value that specifies the unary operation to perform, such as Negate, etc.

6.15.1 Class Summary

```
public abstract class UnaryOperationBinder : DynamicMetaObjectBinder {
    protected UnaryOperationBinder(ExpressionType operation);

    public ExpressionType Operation { get; }

    public abstract DynamicMetaObject FallbackUnaryOperation
        (DynamicMetaObject target, DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackUnaryOperation
        (DynamicMetaObject target);
```

6.15.2 Operation Property

Returns an ExpressionType value representing the specific unary operation to be performed.

6.16 BinaryOperationBinder Abstract Class

Represents a binary operation.

Examples: a + b, a * b

Contains an ExpressionType value that specifies the binary operation to perform, such as Add, Subtract, etc. The left operand’s DynamicMetaObject performs the binding or falls back to the language binder. There is currently no protocol for a call site to ask the right operand to perform a binary operation with a left operand (you can't assume the operation is commutative).

6.16.1 Class Summary

```
public abstract class BinaryOperationBinder : DynamicMetaObjectBinder {
    protected BinaryOperationBinder(ExpressionType operation);

    public ExpressionType Operation { get; }

    public abstract DynamicMetaObject FallbackBinaryOperation
        (DynamicMetaObject target, DynamicMetaObject errorSuggestion);
    public DynamicMetaObject FallbackBinaryOperation
        (DynamicMetaObject target);
```

6.16.2 Operation Property

Returns an ExpressionType value representing the specific binary operation to be performed.

6.17 CallInfo Class

This class represents information about the arguments passed to an invocation binder, such as InvokeMemberBinder, CreateInstanceBinder, or GetIndexBinder.

Note that when you are encoding an InvokeMemberBinder, you should not include the receiver (the implicit “this” argument) as a parameter.

6.17.1 Class Summary

```
public sealed class CallInfo {
    public CallInfo(Int32 argCount, IEnumerable<String> argNames);
    public CallInfo(Int32 argCount, params String[] argNames);

    public int ArgumentCount { get; }
    public ReadOnlyCollection<string> ArgumentNames {get; }
```

6.17.2 ArgumentCount Property

This property returns the total number of argument expressions. For example, the invocation "foo(1, 2, bar=3, baz=4)" has a count of four.

6.17.3 ArgumentNames Property

This property return the names used for any named arguments. If there are N names, then they are the names used in the last N argument expressions. For example "foo(1, 2, bar=3, baz=4)" has a collection of "bar" and "baz".

6.18 BindingRestrictions Class

An instance of the BindingRestrictions class lets a DynamicMetaObject specify the constraints under which its expression would be a valid binding for a given call site in the future. If these constraints are met the next time this call site is hit, the resulting delegate will be found in the cache and rebinding will not occur.

Note that to prevent common mistakes all rules must have restrictions specified, as the vast majority of rule bindings will be applicable in only a subset of situations. If you are certain that a rule you are producing should be applicable every time this call site is reached, you can use

GetExpressionRestriction to produce a conditional “if(true)” test that will be optimized away. Typically, this is not the case, and there would be at least one restriction specified, constraining this binding to a given set of input types.

It is not recommended to implement restrictions that test that a value has a nullable type as nullability is lost when boxing value types to Object. Such restrictions will be modified to test the object against the underlying non-nullable value type and nullability will not be assured.

6.18.1 Class Summary

```
public abstract class BindingRestrictions {
    public static readonly BindingRestrictions Empty;

    public static BindingRestrictions Combine
        (IList<DynamicMetaObject> contributingObjects);
    public static BindingRestrictions GetExpressionRestriction
        (Expression expression);
    public static BindingRestrictions GetInstanceRestriction
        (Expression expression, Object instance);
    public static BindingRestrictions GetTypeRestriction
        (Expression expression, Type type);
    public BindingRestrictions Merge(BindingRestrictions restrictions);
    public Expression ToExpression();
```

6.18.2 GetTypeRestriction Method

Returns a BindingRestrictions instance that validates that the specified expression’s value has the specified type at runtime.

6.18.3 GetInstanceRestriction Method

Returns a BindingRestrictions instance that validates that the specified expression’s value is the exact specified instance at runtime.

6.18.4 GetExpressionRestriction Method

Returns a BindingRestrictions instance that validates that the arbitrary restriction expression specified evaluates to be true at runtime.

6.18.5 Merge Method

Returns a BindingRestrictions instance that combines this instance with the restrictions specified in the argument. This is an AND combination, requiring all restrictions to pass for the rule to match.

6.18.6 Combine Static Method

Returns a BindingRestrictions instance which combines the restrictions from each of a set of DynamicMetaObject instances. This is an AND combination, requiring all restrictions to pass for the rule to match.

6.19 CallSite and CallSite<T> Classes

An instance of CallSite<T> represents a given dynamic call site emitted by a language compiler when it encounters a dynamic expression. Compiling a DynamicExpression node in an expression tree also produces a CallSite<T>.

When emitting a dynamic call site, the compiler must first generate a CallSite<T> object by calling the static method CallSite<T>.Create. The compiler passes in the language-specific binder it wants this call site to use to bind operations at runtime. The T in the CallSite<T> is the delegate type that provides the signature for the site's Target delegate that holds the currently cached rule, and must accept a first parameter of type CallSite.

6.19.1 Class Summary

```
public abstract class CallSite {
    public CallSiteBinder Binder { get; }

    public static CallSite Create
        (Type delegateType, CallSiteBinder binder);
}

public sealed class CallSite<T> : CallSite {
    public T Target;

    public T Update { get; }

    public static CallSite<T> Create
        (CallSiteBinder binder);
}
```

6.19.2 Create Static Method

The static factory method Create on the CallSite<T> class returns a new instance of CallSite<T> for use as a dynamic call site. Compilers that generate assemblies directly as .NET IL (such as the C# compiler) should make sure the user assembly caches these CallSite instances at runtime so they don't need to be regenerated each time this call site is hit. Languages that use Expression Trees as their code generation (languages built on the DLR such as IronPython) can just use DynamicExpressions, letting the Expression Tree compiler do the work.

6.19.3 Target Field

Holds onto the delegate representing the L0 cache for this dynamic call site. This delegate may be invoked with a set of arguments to dispatch the dynamic operation represented by the call site.

6.19.4 Update Property

Returns the delegate to be invoked from within the Target delegate's implementation when there is an L0 cache miss, making it necessary to update the Target delegate.

6.20 StrongBox Class

An instance of StrongBox<T> may be used by binders to represent values passed by reference. The type parameter T represents the type of the value to be referenced.

6.20.1 Class Summary

```
public class StrongBox<T> {
    public T Value;

    public StrongBox<T>(T value);
    public StrongBox<T>();
```

6.21 DynamicObject Class

The simplest way to give your own class custom dynamic dispatch semantics is to derive from the DynamicObject base class. While ExpandoObject only dynamically adds and removes members, DynamicObject lets your objects fully participate in the dynamic object interoperability protocol. There are several abstract operations users of your object can then request of it dynamically, such as getting a member, setting a member, invoking a member on the object, indexing the object, invoking the object itself, or performing standard operations such as addition, multiplication, etc. DynamicObject lets you choose which operations to implement, and lets you do it much more easily than a language implementer.

For more information about DynamicObject, check out the accompanying Getting Started with the DLR for Library Authors on the DLR CodePlex site under Specs and Docs.

6.21.1 Class Summary

```
public class DynamicObject : IDynamicMetaObjectProvider {
    protected DynamicObject();

    public virtual Boolean TryBinaryOperation
        (BinaryOperationBinder binder, Object arg, out Object result);
    public virtual Boolean TryConvert
        (ConvertBinder binder, out Object result);
    public virtual Boolean TryCreateInstance
        (CreateInstanceBinder binder, Object[] args,
         out Object result);
    public virtual Boolean TryDeleteIndex
        (DeleteIndexBinder binder, Object[] indexes);
    public virtual Boolean TryDeleteMember(DeleteMemberBinder binder);
    public virtual Boolean TryGetIndex
        (GetIndexBinder binder, Object[] args, out Object result);
    public virtual Boolean TryGetMember
        (GetMemberBinder binder, out Object result);
    public virtual Boolean TryInvoke
        (InvokeBinder binder, Object[] args, out Object result);
    public virtual Boolean TryInvokeMember
        (InvokeMemberBinder binder, Object[] args, out Object result);
    public virtual Boolean TrySetIndex
        (SetIndexBinder binder, Object[] indexes, Object value);
    public virtual Boolean TrySetMember
        (SetMemberBinder binder, Object value);
    public virtual Boolean TryUnaryOperation
        (UnaryOperationBinder binder, out Object result);

    public virtual DynamicMetaObject GetMetaObject
        (Expression parameter);
```

6.22 ExpandoObject Class

The ExpandoObject class is an efficient implementation of a dynamic property bag provided for you by the DLR. It allows you to dynamically retrieve and set its member values, adding new members per instance as needed at runtime, as typically expected of dynamic objects in languages such as Python and JScript. Instances of ExpandoObject support consumers in writing code that naturally accesses these dynamic members with dot notation (o.foo) as if they were static members, instead of something more heavyweight such as o.GetAttribute("foo").

For more information about ExpandoObject, check out the accompanying Getting Started with the DLR for Library Authors on the DLR CodePlex site under Specs and Docs.

6.22.1 Class Summary

```
public sealed class ExpandoObject : IDynamicMetaObjectProvider,
IDictionary<String,Object>, ICollection<KeyValuePair<String,Object>>,
IEnumerable<KeyValuePair<String,Object>>, IEnumerable {
    public ExpandoObject();
```